Polynomial Inversion Algorithms in Constant Time for Post-Quantum Cryptography

Abhraneel Dutta[1], Emrah Karagoz[1], Edoardo Persichetti[1], and Pakize Sanal[1]

Florida Atlantic University, USA
{adutta2016, ekaragoz2017, epersichetti, psanal2018}@fau.edu

Abstract. The computation of the inverse of a polynomial over a quotient ring or a finite field plays a very important role during the key generation of post-quantum cryptosystems like NTRU, BIKE, and LEDACrypt. It is therefore important that there exist an efficient algorithm capable of running in constant time, to prevent timing side-channel attacks. In this article, we study both constant-time algorithms based on Fermat's Little Theorem and the Extended GCD Algorithm, and provide a detailed comparison in terms of performance. According to our conclusion, we see that the constant-time Extended GCD-based Bernstein-Yang's algorithm shows a better performance with 1.76x-3.76x on `x86` platforms compared to FLT-based methods. Although we report numbers from a software implementation, we additionally provide a short glimpse of some recent results when these two algorithms are implemented on various hardware platforms. Finally, we also explore other exponentiation algorithms that work similarly to the Itoh-Tsuji inversion method. These algorithms perform fewer polynomial multiplications and show a better performance with 1.56x-1.96x on `x86` platform compared to Itoh-Tsuji inversion method.

Keywords: Post-Quantum, Polynomial Inversion, Constant Time, QC-MDPC

1 Introduction

In post-quantum cryptography, constant-time algorithms play a significant role in ensuring the security of a cryptographic system. A constant-time algorithm runs for the same time regardless of the input data size. This ensures that cryptosystems resist timing attacks.

Polynomial inversion not only consumes most of the time and resources among all finite field arithmetic operations, but also plays a crucial role in the key generation of several cryptographic algorithms such as NTRU [9], LEDAcrypt [2], and BIKE [1], which use polynomial inversion to calculate the private key from the public key.

The two main types of polynomial inversion algorithms are derived from the Extended GCD algorithm and Fermat's Little Theorem, where in both cases the inversion is performed by a sequence of multiplications and squarings over a finite field or polynomial quotient ring. In the area of post-quantum cryptography, since they play a very important role in the above-mentioned modern cryptosystems, these two algorithms are usually modified to run in constant time, to provide greater security against timing attacks. In [7], the inversion based on Fermat's Little Theorem [10] is made isochronous, while Bernstein and Yang in [4] provided a constant-time polynomial inversion based on the Extended GCD algorithm.

1.1 Literature, Our Contribution and Paper Organization

In this article, we analyze the performance of both the constant-time algorithms mentioned above, specifically for the BIKE and LEDACrypt setups, based on their mathematical foundations. In both of these cryptosystems, the polynomial inversion is performed over $\mathbb{F}_2[x]/\langle x^p + 1\rangle$ where $x^p + 1 = (x + 1)q(x)$ where $q(x)$ is an irreducible polynomial and p is prime number such that 2 is primitive modulo p. which is the most time consuming operation in the key generation. We compared their performance in terms of the number of field operations, for example, multiplication and squaring.

We focus on Bernstein and Yang's polynomial inversion (BY) [4] and Itoh and Tsujii (IT) [10] in Section 2. While BY inversion is based on Extended GCD-algorithm, IT inversion is based on Fermat's Little Theorem. We make a comparison between these two algorithms based on how many field operations occur during their runtime. We also have a very brief discussion on the recent work on these two algorithms, where the authors have undergone different kinds of hardware implementations. In Section 3, we explore other variants that computationally perform with less number of polynomial multiplications compared to IT inversion. A variant called CEA inversion and proposed by Chang et al. [5] reduces the number of required multiplications by factorizing $m-1$ where polynomial inversion is performed in a finite field $\mathbb{F}_{2^m}$ for some positive integer $m \geq 2$. Another variant called TYT inversion, which was initially proposed by Takagi et al. in [14] and later improved by Chen et al. in [6], significantly reduces the number of polynomial multiplications by breaking down $(m-1)$ into multiple factors along with a minor remainder as $m - 1 = \sum_{i=1}^{k} r_i + h$. Short Addition Chain (SAC) method proposed by [11] improves over other variants by breaking down the value of $(m - 1)$ into multiple components along with a remainder, while ensuring that the remainder is a part of the Short Addition Chain (SAC) of any component within the factors (r_i's) in the decomposition $(m-1) = \sum_{i=1}^{k} r_i + h$. In Section 4, we adapt these algorithms for the prime numbers p used in BIKE and LEDACrypt, as referenced in [1,3]. We evaluate their computational cost and compare the number of polynomial multiplications. Notably, these algorithms can be executed in constant time, as the factoring and decomposition steps are performed on the exponent $p - 2$ when computing $\alpha(x)^{-1} = ((\alpha(x))^{2^{p-2}-1})^2$.

This remains consistent regardless of the degree of the polynomial. Based on our analysis and the results from our software implementation[1], as shown in Table 2, SAC and CEA inversions show a better performance with 1.56x-1.96x on `x86` and 1.24x-1.49x on `arm64` compared to the ITI and TYT methods. However, BY inversion has a better performance with 1.76x-3.76x on `x86` and 1.38x-2.56x on `arm64` compared to IT and its variants. Therefore, it becomes evident that IT variants perform fewer polynomial multiplications while computing the inverse with specific choice for the primes. This improvement can be achieved through an optimized decomposition and factoring setup, especially as the Hamming weight of $p-2$ increases for various primes presented in Table 1. On the other hand, IT variants seem better than BY inversion in the hardware designs by only comparing the number of polynomial multiplications; because of that, the polynomial squaring can be implemented as nearly "cost-free" performance overhead in the hardware designs through special methods. Finally, we provide a conclusion in Section 5 to complete our analysis. Based on our comparison between Bernstein-Yang and IT inversion (with its variants), we can observe that different inversion algorithms have varying performances in terms of the number of multiplications required to compute the inverse of an element $\alpha(x)$ in a ring like $\mathbb{F}_2[x]/\langle x^p+1\rangle$.

1.2 Notation

Since we focus on inversion in BIKE and LEDACrypt setup, the invertible polynomials are chosen from the quotient ring $\mathcal{R} = \mathbb{F}_2[x]/\langle x^p+1\rangle$ where where $x^p+1 = (x+1)q(x)$ where $q(x)$ is an irreducible polynomial and p is prime number such that 2 is primitive modulo p. The degree of a polynomial $\alpha(x)$ is denoted as $\deg(\alpha)$. Polynomials are stored as vectors such as, for a $p-1$-degree polynomial $f(x) = \sum_{i=0}^{p-1} a_i x^i$ is stored as $(a_0, a_1, \ldots, a_{p-1})$ where $a_i \in \mathbb{F}_2$ for all $0 \leq i \leq p-1$. The i-th coefficient is represented as $f[i]$. We also use $\log(\cdot)$ as the base 2 logarithm $\log_2(\cdot)$. The binary representation of an integer a is denoted by $(1a^{(n-2)} \ldots a^{(0)})_2$ with its bit-length $n = |a|$, and least and most significant bits are $a^{(0)}$ and 1 respectively. The Hamming weight of an integer a is defined as the number of 1's in its binary representation and denoted by $\mathrm{wt}(a)$. For a polynomial $\alpha(x) = \sum_{i=0}^{p-1} a_i x^i$ in $\mathcal{R}$ the *support of* $\alpha(x)$ denoted $supp(\alpha)$ is the set of positions of non-zero coefficients. In other words $supp(\alpha) = \{i \in \{0, 1, \ldots, p-1\} |\ a_i \neq 0\}$.

2 Constant-Time Polynomial Inversion Algorithms in PQC

We focus on Bernstein and Yang's [4] polynomial inversion (BY) based on Extended GCD-algorithm, and Itoh and Tsujii's [10] polynomial inversion (IT) based on Fermat's Little Theorem. We then compare these two algorithms based on how many field operations occur during their runtime, and different kinds of their hardware implementations.

[1] Our software is available at `https://github.com/ekaragoz77/polyinv`

2.1 Bernstein-Yang's Constant-Time Polynomial Inversion

Bernstein and Yang proposed a constant-time extended GCD algorithm for polynomials in [4]. The algorithm takes as input two polynomials f and g and outputs the polynomials $\mathrm{GCD}(f, g)$, u and v such that $\mathrm{GCD}(f, g) = u \cdot f + v \cdot g$. Since, during the inversion in BIKE and LEDACrypt setup, an invertible polynomial $\alpha(x)$ is chosen from $\mathcal{R}$, it is evident that $\mathrm{GCD}(x^p + 1, \alpha(x)) = 1$ and the inverse can be obtained as follows:

$$\begin{aligned} 1 &= (x^p + 1) \cdot u(x) + \alpha(x) \cdot v(x) \\ \implies 1 &\equiv \alpha(x) \cdot v(x) \mod x^p + 1 \\ \implies \alpha(x)^{-1} &= v(x) \mod x^p + 1 \end{aligned}$$

Bernstein and Yang introduced a function called *divstep* that performs a polynomial division as a sequence of division steps.

Definition 1. *The divstep function over a field of characteristic* 2 *is defined as*

$$divstep : \mathbb{Z} \times \mathbb{F}_2[x] \times \mathbb{F}_2[x] \to \mathbb{Z} \times \mathbb{F}_2[x] \times \mathbb{F}_2[x]$$

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

Here δ represents the degree of difference between f and g. The output of this function is two polynomials, the first of which is the higher degree of the two input polynomials, and the second of which is derived by subtracting the two input polynomials to remove the constant term. The elimination part, over the finite field of characteristic 2 can be performed by a simple right shift operation after the subtraction is done. In the main inversion algorithm, the divstep function is called multiple times to the input (δ, f, g) that can be expressed as :

$$(\delta_n, f_n, g_n) = divstep^n(\delta, f, g)$$

where δ_n, f_n, g_n are the degree difference and the two polynomials obtained after n-th division steps. The n-th such transition or n number of *divstep* calls for the input polynomials is derived by matrix multiplication. Consider the input polynomials as a vector $(f, g)^T$ and upon one divstep call it gives the transition $(f, g)^T \to (f_1, g_1)^T$ by performing the following matrix multiplication:

$$\begin{pmatrix} f_1 \\ g_1 \end{pmatrix} = T(\delta, f, g) \begin{pmatrix} f \\ g \end{pmatrix}, \text{ where } T(\delta, f, g) = \begin{cases} \begin{pmatrix} 0 & 1 \\ \frac{g(0)}{x} & \frac{-f(0)}{x} \end{pmatrix} \text{ if } \delta > 0 \ \& \ g(0) \neq 1 \\ \begin{pmatrix} 1 & 0 \\ \frac{-g(0)}{x} & \frac{f(0)}{x} \end{pmatrix} \text{ otherwise.} \end{cases}$$

The i-the step transition matrix is represented as $T_i = T(\delta_i, f_i, g_i)$. The n-the degree of iterations of *divstep* function gives

$$\begin{pmatrix} f_n \\ g_n \end{pmatrix} = T_{n-1} \cdots T_0 \cdot \begin{pmatrix} f \\ g \end{pmatrix} = \begin{pmatrix} u_n & v_n \\ q_n & r_n \end{pmatrix} \begin{pmatrix} f \\ g \end{pmatrix}$$

where $\begin{pmatrix} u_n & v_n \\ q_n & r_n \end{pmatrix} = T_{n-1} \cdots T_0$ is the n-th transition matrix for f and g.

From Theorem C.1 of [4], we infer that for two polynomials R_0 and R_1 over $\mathbb{F}_2$, where $d_0 = \deg(R_0)$ and $d_1 = \deg(R_1)$ with $d_0 > d_1 \geq 0$, it takes $2d_0 - 2d_1$ many divsteps to complete the division of R_0 by R_1. The *divstep* function plays the key role in computing the input polynomial's inverse. According to [4, Thm 6.2], the inversion algorithm makes $2d_0 - 1$ *divstep* calls to compute $R_1^{-1} \mod R_0$.

Algorithm Analysis. Bernstein and Yang, in [4], utilized the *divstep* function to handle division steps during the implementation of the Extended Euclidean Algorithm in constant time.

Bernstein and Yang have also introduced a recursive algorithm *jumpdivstep* (Algorithm 2) that reduces the cost of performing n *divteps* where n is large. The algorithm uses a divide-and-conquer rule to calculate δ_n and the n-step transition matrix $T_{n-1} \cdots T_0$. It divides the operands into smaller segments up to a point based on the designer's preference for efficiency. Theorem 4.5 of [4] says that the first t coefficients of f_n and g_n for some non-negative integers n, t determine the transition matrices $T_n, T_{n+1}, \ldots, T_{n+t-1}$; $\delta_{n+1}, \delta_{n+2}, \ldots, \delta_{n+t}$, the first t coefficients of f_{n+1}, first $t-1$ coefficients of (f_{n+2}, g_{n+1}); first $t-2$ coefficients of (f_{n+3}, g_{n+2}) and so on through first coefficient of (f_{n+t}, g_{n+t-1}). The *divstep* algorithm, upon n calls, outputs δ_n the polynomial f_n, g_n with degree $m, m-1$ respectively if $n = 0$ or $m-(n-1)$, $m-n$ respectively if $n \geq 1$ and the transition matrix $T_{n-1}, \cdots, T_0$ where $m = \deg(f)$. The *jumpdivstep* algorithm jumps j steps and computes δ_j and the jth step transition matrix

$$P = T_{j-1} \cdots T_0 = \begin{pmatrix} u_j & v_j \\ q_j & r_j \end{pmatrix}.$$

Next, the algorithm obtains

$$\begin{pmatrix} f_j \\ g_j \end{pmatrix} = \begin{pmatrix} u_j & v_j \\ q_j & r_j \end{pmatrix} \begin{pmatrix} f \\ g \end{pmatrix}.$$

It then takes the first $n - j$ coefficients of (f_j, g_j) according to [4, Thm. 4.5] and performs the remaining $n - j$ *divsteps* by another "jump" obtaining the transition matrix $Q = T_{n-1} \cdots T_j$. Finally, it outputs δ_n and n-th transition matrix $P \times Q = T_{n-1} \cdots T_0$.

According to [4, Thm. 6.2], the main algorithm has to run $2m - 1$ *divsteps* to find $\alpha^{-1}(x)$ for input polynomials $f(x)$ and $\alpha(x)$ with $\deg(f) = m$ and $\mathrm{GCD}(f, \alpha) = 1$. The main algorithm converts the coefficients of these polynomials by reversing them and storing the results as $f(x) = x^m f(1/x)$ and $g(x) = x^{m-1}\alpha(1/x)$. Bernstein and Yang use *jumpdivstep* function and choose

the splitting point to be $j = \lfloor n/2 \rfloor$ as it gives the optimal result according to their observation. After successfully implementing n division steps, the inverse of $a(x)$ is obtained by reversing the coefficients of $v_{2m-1}(x)$.

Algorithm 1: *divstep*

Input: Two polynomials $f(x)$ and $g(x)$, precision value t, δ, and the number of steps n

Output: δ, a matrix $H = T_{n-1} \cdots T_0$, as the product of n transition matrices, so that $\begin{pmatrix} f_n \\ g_n \end{pmatrix} = H \cdot \begin{pmatrix} f \\ g \end{pmatrix}$

$(u, v, q, r) \leftarrow (1, 0, 0, 1)$
$f \leftarrow f.truncate(t)$ and $g \leftarrow g.truncate(t)$
for $j = 1$ **to** n **do**
  $f \leftarrow f.truncate(t)$
  if $\delta > 0$ ***and*** $g(0) \neq 0$ **then**
    $\delta \leftarrow -\delta$
    Swap(f, g), Swap(u, q), Swap(v, r)
  $f_0 \leftarrow f(0)$ and $g_0 \leftarrow g(0)$
  $\delta \leftarrow \delta + 1$
  $g \leftarrow (f_0 \cdot g - g_0 \cdot f)/x$
  $q \leftarrow (f_0 \cdot q - g_0 \cdot u)/x$
  $r \leftarrow (f_0 \cdot r - g_0 \cdot v)/x$
  $g \leftarrow g.truncate(t)$
  $n \leftarrow n - 1$ and $t \leftarrow t - 1$
$H \leftarrow \begin{pmatrix} u & v \\ q & r \end{pmatrix}$
return δ, H

Algorithm 2: *jumpdivstep*

Input: Two polynomials $f(x)$ and $g(x)$, precision value t, δ, and the number of steps n

Output: δ, a matrix $H = T_{n-1} \cdots T_0$, as the product of n transition matrices, so that $\begin{pmatrix} f_n \\ g_n \end{pmatrix} = H \cdot \begin{pmatrix} f \\ g \end{pmatrix}$

if $n \leq w$ **then**
  return $divstep(n, t, \delta, f, g)$
$j \leftarrow \lfloor \frac{n}{2} \rfloor$
$\delta, f_1, g_1, P_1 \leftarrow jumpdivstep(j, j, \delta, f, g)$
$f', g' \leftarrow P_1 \cdot \begin{pmatrix} f \\ g \end{pmatrix}$
$\delta, f_2, g_2, P_2 \leftarrow jumpdivstep(n - j, n - j, \delta, f', g')$
return δ, $P_2 \times P_1$

Application to BIKE. A crucial aspect of the Bernstein-Yang polynomial inversion algorithm is its ability to be executed in constant time for the BIKE and LEDACrypt configuration. The polynomials to be inverted are picked from the polynomial ring $\mathcal{R} = \mathbb{F}_2[x]/\langle x^p + 1 \rangle$ where p is prime such that $2^p \equiv 1 \mod p$ and $x^p + 1 = (x + 1)(x^{p-1} + \cdots + 1)$. The algorithm finds the inverse of $\alpha(x) \mod x^p + 1$ by using $2p - 1$ division steps through the input polynomials $\alpha(x)$ and $x^p + 1$ as input for the *jumpdivstep* algorithm.

Complexity Analysis. Let us first discuss the complexity of the main *divstep* function in terms of the number of field operations.The *divstep* function, in each of its iterations, performs a conditional swap and elimination. While the swapping cost can be considered negligible, we will focus on the elimination phase's cost derived in each iteration. To determine the complexity of *divstep* we rely on Theorem 4.3 and 4.5 in [4].

We know that the *divstep* function computes the n-th transition matrix

$$\begin{pmatrix} u_n & v_n \\ q_n & r_n \end{pmatrix} = T_n = T_{n-1} \cdots T_0$$

for the n-th *divstep* call. During its i-th iteration, while updating (q_i, r_i) it derives $f(0)q_{i-1}(x) - g(0)u_{i-1}(x)$ and $f(0)r_{i-1}(x) - g(0)v_{i-1}(x)$. From theorem 4.3 in [4] we can determine that, at the i-th iteration of *divstep*, as it produces the transition matrix $\begin{pmatrix} u_i & v_i \\ q_i & r_i \end{pmatrix} = T_i = T_{i-1} \cdots T_0$ it has to compute i number of coefficients for both (q_i, r_i). Therefore, for n *divstep* calls it takes $2(1+2+\ldots+n)$ many field operations to produce all the coefficients for both (q_n, r_n). Let us denote this by N_{qr}.

While updating $g(x)$ it performs $f(0)g_{i-1}(x) - g(0)f_{i-1}(x)$ at i-th *divstep* call *i.e.* it eliminates the head coefficients and rescales the polynomial by giving it a simple right shift. Eventually, during its i-th iteration it computes $(f(0)g_{i-1}[1] - g(0)u_{i-1}[1]), (f(0)g_{i-1}[2] - g(0)u_{i-1}[2]), (f(0)g_{i-1}[3] - g(0)u_{i-1}[3]), \ldots$ until $(m-i)$-th coefficient according to Theorem 4.5 in [4]. Therefore, the number of field multiplications and subtractions is $(m+(m-1)+\cdots+(m-n)) = (n+1)m - n(n+1)/2$. Let us denote this as N_g. Therefore, the total number of field operations for n *divstep* calls, is $\mathcal{O}(n(m+kn))$ where k is a constant.

Bernstein and Yang reduced the number of operations in computing n number of *divsteps* by using a divide-and-conquer strategy called "jumps". The algorithm is divided into two halves using *jumpdivstep*, where half of the n *divsteps* is done using half of the coefficients of the input polynomials, followed by updating (f, g) using advanced multiplication (FFT), and then doing the remaining half of steps. The *jumpdivstep* algorithm takes $n, m, f(x), g(x)$ as inputs. So the complexity can be expressed along with the base results as:

$$\begin{aligned} &T(n, m) = 2T(n/2, n/2) + (n+m)\log(n+m) \\ &T(1, m) = m,\ T(n, 1) = 1,\ T(1, 1) = c \ \text{ for some constant } c \end{aligned}$$

Next, we show that the complexity to compute n *divsteps* using *jumpdivstep* along with FFT multiplication takes approximately $(n+m)\log(n+m) + c'n(\log(n))^2$ many operations for sufficiently large n. The result can derived by the following:

$$\begin{aligned} T(n, m) &= 2T(n/2, n/2) + (n+m)\log(n+m) \\ &\approx cn + n((\log(n))^2 + \log(n) - 2) + (n+m)\log(n+m) \\ &\leq (n+m)\log(n+m) + c'n(\log(n))^2 \qquad \text{for sufficiently large } n \end{aligned}$$

Algorithm 3: `PolyInvIT`: Polynomial Inverse by using IT Method

Input: An invertible polynomial $\alpha \in \mathcal{R} = \mathbb{F}_2[x]/\langle x^p + 1\rangle$
Output: $\alpha^{-1} = \alpha^{2^{p-1}-2}$

Precomputation: the binary representation of $p-2 = (e^{(s-1)}e^{(s-2)} \dots e^{(1)}1)_2$
$\delta \leftarrow \alpha$ and $\gamma \leftarrow \alpha$
for $i = 1$ **to** $s-1$ **do**
    $d \leftarrow 2^{i-1}$
    $\delta \leftarrow \delta \cdot \delta^{2^d}$
    if $e^{(i)} = 1$ **then**
        $d \leftarrow 2^i$
        $\gamma \leftarrow \delta \cdot \gamma^{2^d}$
$\delta \leftarrow \gamma^2$
return δ

Therefore, for regular *divstep* it takes $O(n(n+m))$ operations while using *jump-divteps* reduces the number of operations to $O(n(\log n)^2)$.

2.2 Itoh-Tsuji's Polynomial Inversion

The Itoh Tsuji Inversion Algorithm [10], more frequently known as the ITI algorithm, is based on Fermat's Little theorem that is used for polynomial inversion over a finite field $\mathbb{F}_{2^m}$ for some integer $m \geq 1$. In this section, we will discuss the details of a variant of the ITI algorithm presented in [7] which is generalized to be used for the set of polynomial rings that are used in BIKE and in other QC-MDPC-based schemes.

Algorithm Analysis. The constant-time version of ITI algorithm derives the inverse of $\alpha(x) \in \mathcal{R}^*$ with the help of Fermat's Little Theorem as follows:

$$\alpha(x)^{-1} \equiv \alpha(x)^{2^{p-1}-2} \equiv (\alpha(x)^{(2^{p-2}-1)})^2 \mod x^p + 1 \tag{1}$$

Both the ITI and its constant-time version calculate the inverse of the input polynomial using the square-and-multiply strategy in a more efficient way. In the ITI inversion algorithm in [10] the exponent is a power of 2 while for the constant-time algorithm in [7] $p-2$ might not be. Therefore, Drucker, Gueron and Kostic gave a decomposition of $2^{p-2}-1$ for an efficient computation in [7] as follows.
Let $p-2 = \sum_{\substack{k_j \in supp(p-2) \\ 1 \leq j \leq t}} 2^{k_j}$ where $t = |supp(p-2)|$.

$$2^{p-2}-1 = \sum_{i \in supp(p-2)} (2^{2^i}-1)2^{(p-2) \bmod 2^i}$$

From the above decomposition the inversion is computed as:

$$\alpha(x)^{-1} = (\alpha(x)^{2^{p-2}-1})^2 = \Big(\prod_{i \in supp(p-2)} (\alpha(x)^{2^{2^i}-1})^{(p-2) \bmod 2^i} \Big)^2$$
$$= \Big(\prod_{j=1}^{t} (\alpha(x)^{2^{2^{k_j}}-1})^{(p-2) \bmod 2^{k_j}} \Big)^2 = \Big(\prod_{j=1}^{t} (\alpha(x)^{2^{2^{k_j}}-1})^{2^{\sum_{r=j-1}^{1} 2^{k_r}}} \Big)^2.$$

The authors in [7] presented an intelligent approach for calculating the 2^k-th power of the polynomial. First, let us observe the following derivation for a given polynomial $\alpha(x) = \sum_{j=0}^{p-1} a_j x^j \in \mathcal{R}$ with $a_j \in \mathbb{F}_2$

$$\alpha(x)^{2^k} = (\sum_{j=0}^{p-1} a_j x^j)^{2^k} = (\sum_{j \in supp(\alpha)} x^j)^{2^k} = \sum_{j \in supp(\alpha)} (x^j)^{2^k} = \sum_{j \in supp(\alpha)} x^{j \cdot 2^k \bmod p}.$$

Here, we observe that the j-th coefficient of $\alpha(x)$ becomes $(j \cdot 2^k \mod p)$-th coefficient of the polynomial $\alpha(x)^{2^k}$. Therefore, for the positive integer k the sequence of coefficients of $\alpha(x)$ is given a permutation to obtain the sequence of coefficients of $\alpha(x)^{2^k}$ represented by β_i's for $0 \leq i \leq p-1$ as follows:

$$(a_i)_{i=0}^{p-1} = (b_i)_{i=0}^{p-1} \text{ where } b_i = a_{(i \cdot 2^k) \bmod p}$$

Complexity Analysis. In each iteration, the inversion algorithm carries out a required polynomial multiplication and exponentiation, and if $(p-2)_2[i] = 1$, it also performs an additional polynomial multiplication and exponentiation. After all the iterations are implemented it outputs the inverse of the input polynomial multiplication with a final squaring. Therefore the inversion algorithm requires $\lfloor \log(p-2) \rfloor + \mathrm{wt}(p-2) - 1$ polynomial multiplications and $\lfloor \log(p-2) \rfloor + \mathrm{wt}(p-2) - 1$ exponentiation in $\mathcal{R}$. Clearly, the efficiency is determined by both $|p-2|$ and $\mathrm{wt}(p-2)$, selecting values of p that have a smaller $|p-2|$ and $\mathrm{wt}(p-2)$ result in improved performance.

We observed in the last section squaring can be performed by simple permutations as shown in [7,8,13]. To compute $a(x)^{2^k}$ *i.e* the 2^k-th power of $a(x) \in \mathcal{R}$ namely a k-squaring we only have to perform the following permutation:

$$\pi_k(j) : j \to j \cdot 2^k (\mathrm{mod}\ p)$$

for $i \in \{0, 1, \ldots, p-1\}$. The authors in [7] observed that these permutations of the coefficients can be precomputed and stored in a look-up table that can be used for all the relevant values of k that also depend on the public value p. The required storage is $\lfloor \log(p-2) \rfloor + \mathrm{wt}(p-2) + 1$ tables, each of which has $|p|$ values. In total, it needs to store $p \cdot (\lfloor \log(p-2) \rfloor + \mathrm{wt}(p-2) + 1) \cdot \lceil \log(p) \rceil$ bits. This makes the implementation significantly faster as the cost of exponentiation becomes linear. Although this storage requirement is not suitable for microcontrollers as mentioned in [8] but the authors in [3] have optimized

it with smaller look-up tables containing $(\lceil \log(p-2) \rceil - 1)$ values obtained as $2^k \pmod p$ $k \in \{1, 2, \ldots, \lceil \log(p-2) \rceil\}$. At runtime, the j-th coefficient's position in $a(x)^{2^k}$ with $0 \leq j \leq p-1$ is determined by performing a multiplication and modulus operation, specifically $(j \cdot (2^k \pmod p)) \bmod p$.

2.3 Algorithm Comparison for BIKE

Comparison From Mathematical Perspectives As we compare these two algorithms it can be observed that in Algorithm 1 where the most significant part of the calculations take place during the inversion in [4] the division by x for $(f(0)Q - g(0)U)/x, (f(0)R - g(0)V)/x, (f(0)g - g(0)f)/x$ can be done by a simple shift operation since all the polynomials are in binary. So there are no costly operations like polynomial multiplications taking place during the *divtep* calls. The main algorithm, when it uses the *jumpdivtep* function 2 performs only four polynomial multiplications. While on the other side, the ITI variant of polynomial inversion 3 that uses Fermat's Little theorem performs at least $\lfloor \log(p-2) \rfloor + \mathrm{wt}(p-2) + 1$ many polynomial multiplications in $\mathcal{R}$ and if it does not use the lookup table as suggested in [7] it has to perform $\lfloor \log(p-2) \rfloor + \mathrm{wt}(p-2) + 1$ many exponentiations on the top of that. Therefore, constructing the algorithm with the Extended Euclidean algorithm in constant time seems to be more efficient in terms of time complexity or the number of field operations.

Inversions in Recent BIKE Implementations Richter-Brockmann, Mono, and Güneysu in [13] efficiently implemented the BIKE algorithm on FPGAs. They proposed new strategies to compute the polynomial inversion based on Itoh-Tsujii's algorithm [10] that uses Fermat's Little Theorem. They introduced a scaling parameter b and divide the polynomials into chunks of b bits that are processed through different levels of parallelization for inversion. They did not follow the proposed algorithm in [7] as its hardware performance is not remarkable. Although their number of polynomial multiplications is the same as in [7] but the number of squaring is different. For polynomial multiplications, they have used schoolbook based multiplication. For an arbitrary t, the exponentiation of the input polynomial in algorithm 5 in [13] to a power of 2^t can be performed using a squaring chain consisting of $\lfloor t/k \rfloor$ k-squaring and $t \bmod k$ single squaring where $k < t$. For a 2^k exponentiation the authors use the coefficient permutation $\pi_k : i \to i \cdot 2^k \pmod p$. For a fixed k this process, after its initial phase, accomplishes the squaring in $\lceil p/b \rceil$ clock cycles. While with an arbitrary k value, more significantly used, for a larger value of t finishes the squaring within p clock cycles with their squaring module given in Fig. 3 in [13].

Richter-Brockmann, Chen, Ghosh, and Güneysu in [12], in their FPGA implementation of BIKE, optimized the polynomial inversion based on constant-time polynomial inversion in [4] achieving 5.5 times faster key generation compared to previous implementations based on Fermat's Little Theorem. They

present a polynomial multiplier that is optimized to take advantage of the sparsity of QC-MDPC codes. This multiplier is capable of performing all the necessary multiplications in the BIKE algorithm within constant time. They also introduce a constant-time polynomial inversion algorithm based on EEA adapting it from [4]. They split the *divstep* function into two different operations and provide the respective algorithms (Algorithm 5 and 6 in [12]). One calculates the control bits based on δ and the coefficients of f, g while the other one uses these control bits to update the polynomials in Algorithm 4 in [12]. Furthermore, the primary inversion algorithm, namely Algorithm 4 in [12], includes an additional input parameter denoted by s, which regulates the step size and enables the algorithm to execute s *divsteps* during each iteration.

The authors of [12] have contrasted their inversion module with the method in [13] that relies on Fermat's Little Theorem. The design of the multiplier in [13] takes into account the use of a schoolbook-based multiplication and requires a total of $L_{school} = \lceil p/b \rceil \cdot (\lceil p/b \rceil \cdot +3) + 1$ clock cycles to complete. The total number of clock cycles achieved in [13] is the following:

$$\log(p) \cdot (p + L_{school}) + |p| \cdot (\left\lceil \frac{p}{b} \right\rceil + L_{school})$$

In contrast, the latency resulting from the hardware design presented in [12] is determined by the following

$$\lambda \cdot (3 + \left\lceil \frac{s}{d} \right\rceil + \left\lceil \frac{s}{u} \right\rceil + \left\lceil \frac{p}{b} \right\rceil) + \rho + \left\lceil \frac{s}{d} \right\rceil + \left\lceil \frac{p}{b} \right\rceil + 3 \cdot \left\lceil \frac{p}{b} \right\rceil + 13$$

where the parameters are as follows:

- s : Step size *i.e* proceed s *divsteps* in a single iteration
- d : Algorithm 5 in [12] that generates control bits for conditional steps of *divstep* function executes d iterations in one clock cycle.
- u : The submodule consisting of shifts, addition, and multiplexing operations of the hardware design presented in Figure 6 in [12] completes its computation with s consecutive basic blocks.
- (λ, ρ) : $\rho = \left\lceil \frac{2p-1-\lambda s}{u} \right\rceil$ where $\lambda = \lceil \frac{2p-1}{s} \rceil$

Table 4 in [12] displays the results of the author's implementation, which achieved a better runtime in terms of clock cycles when compared to the inversion technique introduced in [13].

Recently, an efficient and scalable hardware design for binary polynomial inversion, which has been validated for BIKE implementation, was proposed by Galimberti, Montanaro, and Zoni in [8]. Their inversion design is based on Fermat's Little Theorem. The efficiency of their design is achieved by providing a parallel architecture for exponentiation and multiplication and optimal hardware scheduling. In their design, most significantly, they have introduced modules termed "Mul" and "Exp" that can be used as the resources for multiplication and exponentiation respectively. Their proposed architecture aims to optimize the scheduling of exponentiation and multiplication by utilizing the

"Exp" and "Mul" modules concurrently, whenever feasible. This scheduling optimization reduces the number of operations performed when $(p-2)_2[i] = 1$. To perform an iteration, in their design, the execution time needed is twice that of the longest operation between exponentiation and multiplication, rather than the combined execution time of two exponentiations and two multiplications. The time complexity T_{inv} that they have presented in terms of the number of multiplications and exponentiations is the following

$$\begin{aligned} T_{inv} =& ((2 \cdot (\mathrm{wt}(p-2)-1))-1) \cdot \max\{T_{exp}, T_{mul}\} \\ &+ (zeroes(p-2)+1) \cdot (T_{exp}+T_{mul}) + T_{exp} \end{aligned}$$

where $zeroes(p-2)$ represents number of zeros in the binary representation of $(p-2)$. The authors in [8] have adopted a multiplication architecture from [15], which is both efficient and scalable while they proposed a new exponentiation architecture illustrated in Fig 5 in [8]. The time complexity of multiplication and exponentiation design is represented as T_{mul} and T_{exp} respectively.

3 ITI Variants

In this section, we will be exploring some exponentiation algorithms that are capable to perform fewer number of polynomial multiplications towards the calculation of the inverse of the input polynomial. More importantly, since these algorithms involve factoring and decomposing of the exponents, they can be implemented irrespective of the degree of the input polynomials giving it a constant-time structure. We will be discussing these algorithms in chronological order to observe the improvements over the IT inversion algorithm analyzed in Section 2.2.

3.1 CEA Algorithm (Factoring Method)

The CEA algorithm given by Chang et. al. in [5] involves factoring of the exponent $m-1$ while computing $\alpha(x)^{-1} = ((\alpha(x))^{2^{m-1}-1})^2$ in $\mathbb{F}_{2^m}$. We have adapted their method and claim that it can potentially improve the performance of contant-time IT inversion algorithm in BIKE/LEDACrypt setup.

Let $p-2 = a \cdot b$. Note that both a, b are odd. Now we can factor $2^{p-2}-1$ followed by the exponentiation as

$$\begin{aligned} 2^{p-2}-1 = 2^{a \cdot b}-1 &= (2^a-1)((2^a)^{b-1} + (2^a)^{b-2} + \ldots + 2^a + 1) \\ \Rightarrow \alpha^{2^{p-2}-1} &= (\alpha^{2^a-1})^{(2^a)^{b-1}+(2^a)^{b-2}+\ldots+2^a+1} \end{aligned}$$

Computing $\beta = \alpha^{2^a-1}$ needs $\lfloor \log(a) \rfloor + \mathrm{wt}(a) - 1$ many polynomial multiplications and $\lfloor \log(a) \rfloor + \mathrm{wt}(a) - 1$ squaring in $\mathcal{R}$ that can be performed by coefficient permutations. Computing β^t where $t = 1 + 2^a + \ldots + (2^a)^{b-2} + (2^a)^{b-1}$ needs $(\lfloor \log_2(b) \rfloor + \mathrm{wt}(b) - 1)$ many polynomial multiplications (Lemma 2.1 in [6]). Therefore, the total number of polynomial multiplications is $(\lfloor \log_2(b) \rfloor + \mathrm{wt}(b) - 1) + (\lfloor \log_2(a) \rfloor + \mathrm{wt}(a) - 1)$.

Algorithm 4: `PolyInvCEA`: Polynomial Inverse by using CEA Method

Input: An invertible polynomial $\alpha \in \mathcal{R} = \mathbb{F}_2[x]/\langle x^p + 1\rangle$
Output: $\alpha^{-1} = \alpha^{2^{p-1}-2}$

Precomputation of integers a and b such that $p - 2 = a \cdot b$
Precomputation of the binary representation of $a = (1a^{(s-2)} \dots a^{(0)})_2$
Precomputation of the binary representation of $b = (1b^{(t-2)} \dots b^{(0)})_2$
$\gamma \leftarrow \alpha$
for $i = s - 2$ **to** 0 **do**
    $d \leftarrow 2^i$
    $\gamma \leftarrow \gamma \cdot \gamma^{2^d}$
    if $a^{(i)} = 1$ **then**
        $\gamma \leftarrow \alpha \cdot \gamma^{2^d}$
$\delta \leftarrow \gamma^2$
for $i = t - 2$ **to** 0 **do**
    $d \leftarrow a \cdot 2^i$
    $\delta \leftarrow \delta \cdot \delta^{2^d}$
    if $b^{(i)} = 1$ **then**
        $\delta \leftarrow \gamma \cdot \delta^{2^d}$
return δ

3.2 TYT Algorithm (Decomposition Method)

Takagi, Yoshiki, and Takagi proposed an algorithm in that enhances the efficiency of computing multiplicative inverses in $\mathbb{F}_{2^m}$, which is pivotal in cryptographic and error-correcting code applications. This algorithm builds on previous methods, such as those by Itoh and Tsujii [14] and Chang et al. [5], by reducing the number of required multiplications through a strategic decomposition of $m - 1$ into multiple factors and a small remainder. It offers improved performance for certain values of $m - 1$, demonstrating significant reductions in computational complexity and making the process of multiplicative inversion more efficient and practical for real-world applications.

We can adapt their algorithm and decompose the exponent $p - 2$, which potentially can help us reduce the number of polynomial multiplications for some specific values of p. Let $p - 2 = \prod_{i=1}^{k} r_i + h$ and we can deduce the following equation:

$$2^{p-1} - 2 = 2^{p-2} + 2^{p-2} + \cdots + 2^{p-h-1} + 2^{p-h-1} - 2$$
$$\alpha^{-1} = \alpha^{2^p-2} = \underbrace{\alpha^{2^{p-2}} \cdot \alpha^{2^{p-2}} \cdots \alpha^{2^{p-h-1}} \cdot \alpha^{2^{p-h-1}}}_{h \text{ mults}} \cdot (\alpha^{2^{p-h-2}-1} \cdot)^2$$

The number of multiplications to compute $\alpha^{2^{p-h-2}-1} = \alpha^{2^{\prod_{i=1}^{k} r_i}}$ can be deduced by the methods discussed in the Section3.1. Therefore, it needs $\sum_{i=1}^{k}(\lfloor \log(r_i) \rfloor +$

$\mathrm{wt}(r_1) - 1$ many multiplications and the total number of polynomial multiplications is $\sum_{i=1}^{k}(\lfloor \log(r_i) \rfloor + \mathrm{wt}(r_1) - 1) + h$.

Y. Li, G. Chen, Y. Chen and J. Li proposed an algorithm that minimizes the number of necessary multiplications by breaking down into multiple factors along with a remainder and reusing intermediate computational results in $\mathbb{F}_{2^m}$ [6], further reducing the number of required multiplications by decomposing $m-1$ into several factors plus a small remainder. We can again use their method in BIKE/LEDACrypt setup to achieve fewer polynomial multiplications with an optimized choice of decomposition of $p-2$. The decomposition of the exponent as per it is in [6] as follows

$$2^{p-1} - 2 = 2^{p-h-1}(2^h - 1) + 2(2^{p-h-2} - 1) \text{ with } p - 2 = \sum_{i=1}^{k} r_i + h$$

$$\alpha^{-1} = \underbrace{(\alpha^{2^h-1})^{2^{m-h}}}_{w(h) \text{ mults}} \cdot \underbrace{(\cdots((\alpha^{2^{r_1}-1}))^{s_{r_2}})^{s_{r_3}}\cdots)^{s_{r_k}}}_{\sum_{i=1}^{k}(\lfloor \log_2(r_i) \rfloor + \mathrm{wt}(r_i) - 1) \text{ mults}}$$

with $s_{r_i} = (2^{a_i-1})^{r_i-1} + (2^{a_i-1})^{r_i-2} \cdots + (2^{a_i-1}) + 1$ with $2 \leq i \leq k$. The decomposition is deduced with $h < r_1$ w.l.o.g. Based on this decomposition, $\alpha^{2^{r_1}-1}$ and α^{2^h-1} can be computed with $\log(r_1) + \mathrm{wt}(r_1) - 1 + \mathrm{wt}(h) - 1$ many multiplications. While the number of multiplications corresponding to the rest of the factors is $\sum_{i=1}^{k}(\lfloor \log_2(r_i) \rfloor + \mathrm{wt}(r_i) - 1)$ followed by a final multiplication. Therefore, the total number of multiplications is $\sum_{i=1}^{k}(\lfloor \log_2(r_i) \rfloor + \mathrm{wt}(r_i) - 1) + w(h)$.

Algorithm 6: `Factorize`

Input: An integer t
Output: A list B as factors of t with minimal multiplication cost m

$B \leftarrow [t]$
if $\mathrm{wt}(t) \leq 2$ ***or*** t *is prime* **then**
  return B
$m \leftarrow \lfloor \log_2(t) \rfloor + \mathrm{wt}(t-1)$
for $i = 3$ **to** $\lfloor \sqrt{t} \rfloor + 1$ **do**
  if $t \bmod i = 0$ **then**
    $q \leftarrow \lfloor t/i \rfloor$
    $c \leftarrow \lfloor \log_2(i) \rfloor + \mathrm{wt}(i) + \lfloor \log_2(q) \rfloor + \mathrm{wt}(q-2)$
    if $c < m$ **then**
      $m \leftarrow c$
      $B \leftarrow \mathrm{Factorize}(i) + \mathrm{Factorize}(q)$
return B

Algorithm 7: `OptDecomp`

Input: An integer n
Output: A list B of integers as a decomposition of n with minimal cost m

$B \leftarrow [n]$
if $\mathrm{wt}(n) \leq 2$ **then**
  return B
$m \leftarrow \lfloor \log_2(n) \rfloor + \mathrm{wt}(n-1)$
for $i = 0$ **to** $\lceil \frac{n}{4} \rceil$ **do**
  if $\mathrm{wt}(i) < \mathrm{wt}(n-2)$ **then**
    $D \leftarrow \mathrm{Factorize}(n-i)$
    $c \leftarrow \mathrm{wt}(i) + \sum_{d \in D}(\lfloor \log_2(d) \rfloor + \mathrm{wt}(d-1))$
    if $c < m$ **then**
      $m \leftarrow c$
      $B \leftarrow D + [i]$
return B

Algorithm 5: `PolyInvTYT`: Polynomial Inverse by using TYT Method

Input: An invertible polynomial $\alpha \in \mathcal{R} = \mathbb{F}_2[x]/\langle x^p + 1\rangle$

Output: $\alpha^{-1} = \alpha^{2^{p-1}-2}$

Precomputation of integers $r_1, \ldots, r_k$ and h s.t. $p - 2 = \prod_{i=1}^{k} r_i + h$

Precomputation of binary representations of $r_1, \ldots, r_k$ and h with bit-lengths $q_1 = |r_1|, \ldots, q_k = |r_k|$ and $\ell = |h|$

$q \leftarrow max(q_1, \ldots, q_k)$

`/* A list` F `of polynomials */`

$F \leftarrow [f_0 = \alpha, f_1 = 0, \ldots, f_{q-1} = 0]$ of length q

for $i = 1$ **to** $q_1 - 1$ **do**

  $d \leftarrow 2^{i-1}$

  $f_i \leftarrow f_{i-1} \cdot (f_{i-1})^{2^d}$

$\delta \leftarrow f_{q_1-1}$

for $i = q_1 - 2$ **to** 0 **do**

  $d \leftarrow 2^i$

  if $r_1^{(i)} = 1$ **then**

    $\delta \leftarrow f_i \cdot \delta^{2^d}$

`/* Compute` γ `from` h `*/`

$\gamma \leftarrow f_{\ell-1}$

for $i = l - 2$ **to** 0 **do**

  $d \leftarrow 2^i$

  if $h^{(i)} = 1$ **then**

    $\gamma \leftarrow f_i \cdot \gamma^{2^d}$

`/* Update` δ `and` F `with` $r_2, \ldots, r_k$ `*/`

$n \leftarrow r_1$ and $f_0 \leftarrow \delta$

for $j = 1$ **to** k **do**

  for $i = 1$ **to** $q_j - 1$ **do**

    $d \leftarrow n \cdot 2^{i-1}$

    $f_i \leftarrow f_{i-1} \cdot (f_{i-1})^{2^d}$

  $\delta \leftarrow f_{q_j-1}$

  for $i = q_j - 2$ **to** 0 **do**

    $d \leftarrow n \cdot 2^i$

    if $r_j^{(i)} = 1$ **then**

      $\delta \leftarrow f_i \cdot \delta^{2^d}$

  $n \leftarrow n \cdot r_j$

$d \leftarrow 2^{r-2-h}$

$\delta \leftarrow \left(\delta \cdot \gamma^d\right)^2$

return δ

3.3 Addition Chain Method

A proposed inversion algorithm in [11] by Mahmoud includes the use of Fermat's Little Theorem and normal basis representation to optimize finite field inversion in $\mathbb{F}_{2^m}$. It also involves decomposing $m-1$ into several factors and a remainder h, where h is chosen from the shortest addition chain (SAC) of one of these factors, effectively minimizing the required multiplications. We can adopt their method to perform better for polynomial inversion than IT inversion in $\mathbb{F}_2[x]/\langle x^p + 1\rangle$. We find the optimized decomposition of $p - 2 = \sum_{j=1}^{k} r_j + h$ where h is within the short addition chain of one of the carefully chosen factors r_i. To discuss this further, we begin with the definition of an addition chain.

Definition 2. *The Short Addition Chain of a positive integer r, denoted as $\mathcal{C}_r$, is a short chain (sequence) of elements (integers) of length n, with the property that r (the last chain element) is obtained by the gradual addition of the previous elements within the chain(or the gradual addition of previous chain elements).*

Algorithm 8: `PolyInvSAC`: Polynomial Inverse by using SAC Method

Input: A polynomial $\alpha \in \mathcal{R} = \mathbb{F}_2[x]/\langle x^p + 1\rangle$

Output: $\alpha^{-1} = \alpha^{2^{p-1}-2}$

`/* Precomputations from SAC Decomposition of` $p-2$ `*/`

Integers r, n and h from the SAC Decomposition of $p-2 = r \cdot n + h$

Addition Chain $C = \{c_0 = 1, c_1, \ldots, c_{t-1} = r\}$

A list of index pairs $A = \{(i_1, i_2) : c_{i_1} + c_{i_2} = c_i \text{ for all } i = 1, \ldots, t-1\}$

Binary representation of $n = (1n^{(k-2)} \ldots n^{(0)})$

Index i_h such that $h = c_{i_h}$

`/* Compute` $\delta_r = \alpha^{(2^r-1)}$ `and */`

`/*` $\delta_h = \alpha^{(2^h-1)}$ `*/`

$L = [\alpha]$

for $i = 1$ **to** $t-1$ **do**

  $c_i \leftarrow C[i]$ and $i_1, i_2 \leftarrow A[i]$ and $c_{i_2} \leftarrow C[i_2]$

  $d \leftarrow 2^{c_{i_2}}$

  Append $L[i_2] \cdot (L[i_1])^d$ to L

$\delta_r \leftarrow L[t-1]$

$\delta_h \leftarrow L[i_h]$

`/* Compute` $\gamma = \delta_r^{2^{(r \cdot n)}-1}$ `*/`

$\gamma \leftarrow \delta_r$

for $i = k-2$ **to** 0 **do**

  $\gamma \leftarrow \gamma \cdot \gamma^{2^{r \cdot 2^i}}$

  if $n^{(i)} = 1$ **then**

    $\gamma \leftarrow \delta_r \cdot \gamma^{2^{r \cdot 2^i}}$

if $h = 0$ **then**

  return γ

else

  $\delta \leftarrow \delta_h \cdot \gamma^{2^h}$

  $\delta \leftarrow \delta^2$

  return δ

The decomposition of $2^{p-1} - 2$ can be done as:

$$2^{p-1} - 2 = 2(2^h \cdot (2^{\sum_{i=1}^{k} r_i} - 1) + (2^h - 1)) = 2((2^{r_1} - 1) \cdot e \cdot 2^h \cdot (2^h - 1))$$

$$\alpha^{-1} = ((\alpha^{2^{r_1}-1})^{(e)2^h} \cdot (\alpha^{2^h-1}))^2 \tag{2}$$

where $e = (((2^{r_1})^{r_2-1} + \cdots + 1) \cdots ((2^{r_1 \cdot r_2 \cdots r_{k-1}})^{r_k-1} + \cdots + 1))$.

To compute α^{-1} as an exponentiation in Equation 2 the algorithm needs at most $\sum_{j=1}^{k} \lfloor (\log_2(r_j) \rfloor + \mathrm{wt}(r_j) - 1) + 1$ many multiplications according to [11, Thm. 5]. Given the decomposition of $p-2 = \sum_{i=1}^{k} r_i + h$, a short addition chain of r_1, namely C_{r_1}, can be constructed with $h \in C_{r_1}$. This guarantees the computation of α^{2^h-1} while calculating $\alpha^{2^{r_1}-1}$. For a given addition chain of $\mathcal{C}_{r_1} = \{c_0, c_1, \ldots, c_{n-1}\}$, a set of pairs $\mathcal{A}_{r_1} = \{(c_1^1, c_1^2), (c_2^1, c_2^2), \ldots, (c_{n-1}^1, c_{n-1}^2) |\ c_i^1 + c_i^2 = c_i \text{ and } c_i^j \in \mathcal{C}_{r_1} \forall\ i = 1, 2, \ldots, n-1 \text{ and } j = 1, 2\}$ is constructed that helps computing $\alpha^{2^{r_1}-1}$ with $n-1$ multiplications as follows:

$$\begin{aligned}
&(\alpha^{2^{c_1^1}-1})^{2^{c_1^2}} \cdot (\alpha^{2^{c_1^2}-1}) = (\alpha^{2^{c_1^1+c_1^2}-1}) = (\alpha^{2^{c_1}-1}) && \text{1 mult}\\
&(\alpha^{2^{c_2^1}-1})^{2^{c_2^2}} \cdot (\alpha^{2^{c_2^2}-1}) = (\alpha^{2^{c_2^1+c_2^2}-1}) = (\alpha^{2^{c_2}-1}) && \text{1 mult}\\
&\quad\vdots \qquad\qquad \vdots \qquad\qquad \vdots \qquad\qquad \vdots\\
&(\alpha^{2^{c_{n-1}^1}-1})^{2^{c_{n-1}^2}} \cdot (\alpha^{2^{c_{n-1}^2}-1}) = (\alpha^{2^{c_{n-1}^1+c_{n-1}^2}-1}) = \boldsymbol{(\alpha^{2^{r_1}-1})} && \text{1 mult}
\end{aligned}$$

Table 1. Comparison of the number of multiplications with different inversion algorithms discussed in this article using primes corresponding to different levels of BIKE implementation from [7].

p	wt$(p-2)$	CEA Factorization	TYT Decomposition	SAC Decomposition	# Mults (ITI)	# Mults (CEA)	# Mults (TYT)	# Mults (SAC)
10499	4	3×3499	$41 \times 256 + 1$	$41 \times 2^8 + 1$	16	20	16	16
12323	4	$3^2 \times 37^2$	$12289 + 32$	$48 \times 2^8 + 33$	16	19	16	16
24781	7	71×349	$3 \times 8257 + 8$	$193 \times 2^7 + 75$	20	22	18	19
27067	9	5×5413	$67 \times 403 + 64$	$211 \times 2^7 + 57$	22	20	21	20
24659	5	3×8219	$5 \times 4112 + 4097$	$385 \times 2^6 + 17$	18	19	18	18
27581	11	3×9193	$163 \times 169 + 32$	$215 \times 2^7 + 59$	24	22	21	20
40973	5	3×13657	$10 \times 4097 + 1$	$20 \times 2^1 1 + 11$	19	22	18	18

4 Experimental Evaluation

The polynomial inversion algorithms are implemented in C and benchmarked on `x86` (AMD Ryzen 5800 @ 3.4 GHz) and `arm64` (Apple Mac Pro M2 @ 2.4 GHz) architectures. The code is compiled using GCC 9.4.0 with the `march=native` and `-O3` optimization flags. The benchmarking results, shown in Table 2, present the median clock cycles (in millions) in both architectures for different prime numbers, as well as the number of function calls for high-level modular polynomial multiplication (`gf2x_mod_mul`) and squaring (`gf2x_mod_sqr`)) functions. Based on that the polynomials are represented in 64-bit blocks, the block multiplications (`mul64`) and squaring (`sqr64`) are performed through *carry-less multiplication instructions* (CLMUL) in both architectures.

Although the polynomial squaring can be implemented as nearly "cost-free" performance overhead in the hardware designs through special bit operations and permutations for the specific target prime, it is better to use block squaring through *carry-less multiplication instructions* (CLMUL) in the software implementations for any target prime rather than using permutations with costly look-up tables. In addition, while a polynomial multiplication operation takes $\mathcal{O}(n^2)$ block multiplications (or $\mathcal{O}(n^{1+\epsilon})$ with special methods like Karatsuba or Toom-Cook) with many XOR operations, a polynomial squaring takes only $\mathcal{O}(n)$ block squarings with fewer XOR operations for modular reduction only.

In Table 2, the SAC and CEA methods show a better performance with 1.56x-1.96x on `x86` and 1.24x-1.49x on `arm64` compared to the ITI and TYT methods. However, the BYI method has a better performance with 1.76x-3.76x on `x86` and 1.38x-2.56x on `arm64` compared to all FLT-based methods. The main advantage of the BYI method over FLT-based methods is the lower number of block multiplications, especially on the low levels of "jumpdivstep", while the number of block multiplications is higher (and fixed) in each polynomial modular multiplication and squaring. On the other hand, for a possible hardware design, FLT-based methods seem better than BY method by only comparing the number of block multiplications.

Table 2. Benchmarking of inversion algorithms, in millions of clock cycles for `x86` and `arm64` architectures (the first two rows for each prime), in the number of function calls for modular polynomial multiplication and squaring (the third and fourth rows for each prime), and in the number of block multiplication and squaring (last two rows for each prime).

p (size64)		**BYI**	**ITI**	**CEA**	**TYT**	**SAC**
	x86- med (Mcc)	**4.35**	**12.85**	**8.38**	**15.02**	**7.66**
10499	arm64- med (Mcc)	**7.37**	**16.32**	**13.78**	**17.35**	**12.02**
(165)	# `gf2x_mod_mul`	N/A	16	20	16	16
	# `gf2x_mod_sqr`	N/A	18,688	10,497	20,992	10,538
	# `mul64`	≈ 1.10M	≈ 0.44M	≈ 0.54M	≈ 0.44M	≈ 0.44M
	# `sqr64`	-	≈ 3.10M	≈ 1.74M	≈ 3.48M	≈ 1.75M
	x86- med (Mcc)	**5.79**	**17.02**	**10.89**	**20.07**	**10.93**
12323	arm64- med (Mcc)	**12.23**	**21.91**	**18.74**	**25.25**	**16.92**
(193)	# `gf2x_mod_mul`	N/A	16	19	16	16
	# `gf2x_mod_sqr`	N/A	20,152	12,321	24,578	12,369
	# `mul64`	≈ 1.51M	≈ 0.60M	≈ 0.71M	≈ 0.60M	≈ 0.60M
	# `sqr64`	-	≈ 3.98M	≈ 2.39M	≈ 4.77M	≈ 2.40M
	x86- med (Mcc)	**21.03**	**66.25**	**42.51**	**64.32**	**42.61**
24659	arm64- med (Mcc)	**45.73**	**95.18**	**77.74**	**96.50**	**77.88**
(386)	# `gf2x_mod_mul`	N/A	18	19	18	19
	# `gf2x_mod_sqr`	N/A	41,040	24,657	41,121	24,739
	# `mul64`	≈ 6.05M	≈ 2.68M	≈ 2.83M	≈ 2.68M	≈ 2.83M
	# `sqr64`	-	≈ 15.88M	≈ 9.54M	≈ 15.91M	≈ 9.57M
	x86- med (Mcc)	**22.11**	**65.64**	**44.13**	**76.33**	**42.18**
24781	arm64- med (Mcc)	**46.30**	**101.52**	**86.50**	**107.21**	**80.28**
(388)	# `gf2x_mod_mul`	N/A	20	22	18	19
	# `gf2x_mod_sqr`	N/A	41,162	24,779	49,542	25,091
	# `mul64`	≈ 6.10M	≈ 3.01M	≈ 3.31M	≈ 2.71M	≈ 2.86M
	# `sqr64`	-	≈ 16.01M	≈ 9.64M	≈ 19.27M	≈ 9.76M
	x86- med (Mcc)	**24.43**	**76.65**	**50.19**	**91.81**	**51.94**
27067	arm64- med (Mcc)	**54.42**	**125.42**	**98.09**	**137.71**	**101.62**
(423)	# `gf2x_mod_mul`	N/A	22	20	21	20
	# `gf2x_mod_sqr`	N/A	43,448	27,065	54,002	27,337
	# `mul64`	≈ 7.09M	≈ 3.94M	≈ 3.58M	≈ 3.76M	≈ 3.58M
	# `sqr64`	-	≈ 18.42M	≈ 11.48M	≈ 22.90M	≈ 11.59M
	x86- med (Mcc)	**25.34**	**81.5**	**53.4**	**94.94**	**53.4**
27581	arm64- med (Mcc)	**57.34**	**139.49**	**111.14**	**146.93**	**103.03**
(431)	# `gf2x_mod_sqr`	N/A	43,962	27,579	55,094	27,850
	# `gf2x_mod_mul`	N/A	24	22	21	20
	# `mul64`	≈ 7.34M	≈ 4.46M	≈ 4.09M	≈ 3.90M	≈ 3.72M
	# `sqr64`	-	≈ 18.99M	≈ 11.91M	≈ 23.80M	≈ 12.03M
	x86- med (Mcc)	**58.14**	**191.2**	**116.62**	**208.12**	**113.43**
40973	arm64- med (Mcc)	**127.29**	**284.38**	**246.86**	**300.02**	**217.01**
(641)	# `gf2x_mod_mul`	N/A	19	22	18	18
	# `gf2x_mod_sqr`	N/A	73,738	40,971	81,940	40,983
	# `mul64`	≈ 17.02M	≈ 7.81M	≈ 9.04M	≈ 7.40M	≈ 7.40M
	# `sqr64`	-	≈ 47.34M	≈ 26.30M	≈ 52.61M	≈ 26.31M

5 Conclusions and Future Work

In this work, from Section 2.1 through Section 2.3, we compared both the constant-time algorithms, and based on their mathematical foundations, we observe that the polynomial inversion by using the Bernstein-Yang algorithm performs fewer number of polynomial multiplications. Although during the hardware implementations of these two algorithms in BIKE and LEDAcrypt setup, the choice depends on specific requirements, and thus they perform accordingly. In [3] the authors claimed to get the best result with the optimized version of Fermat's Little Theorem method using look-up tables on platforms providing AVX2 ISA extensions. On the other hand, the Bernstein-Yang modular inversion over $\mathbb{F}_2$ performs better when both of them were implemented over FPGA for BIKE [12, 13]. Therefore, as a conclusive remark, we can say in general $i.e$ without a particular ring structure like $\mathbb{F}_2[x]/\langle x^p+1\rangle$ the Bernstein Yang modular inversion using the Extended GCD Algorithm is more efficiently performing constant-time algorithm compared to the ITI variant inversion based on Fermat's Little Theorem.

On the other hand, we show that applying ITI variant algorithms in Section 3 to the primes used in BIKE can potentially perform less number of polynomial multiplications as shown in Section 4 during the key generation phase of BIKE. Table 1 suggests that some of the primes perform better compared to the constant-time IT inversion algorithm discussed in Section 2.2. As shown in Table 2, we see that SAC and CEA methods show a better performance with 1.56x-1.96x on `x86` compared to ITI and TYT methods, while BY method show a better performance with 1.76x-3.76x on `x86` compared to all FLT-based methods.

As of now, the hardware implementations indicate that Bernstein-Yang polynomial inversion performs better on FPGA platforms. However, with the results in this article, we believe that ITI variant polynomial inversion algorithms can potentially perform better in hardware implementations with less number of multiplications. Consequently, in the future, a comprehensive hardware implementation of these algorithms will serve as an intriguing next step as a performance improvement in the key generation phase of BIKE and LEDAcrypt.

References

1. Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Santosh Ghosh, Shay Gueron, Tim Güneysu, et al. "BIKE: bit flipping key encapsulation", 2022.
2. Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. "LEDAcrypt: QC-LDPC Code-Based Cryptosystems with Bounded Decryption Failure Rate". *International Workshop on Code-Based Cryptography*, 2019.
3. A. Barenghi and G. Pelosi. "A comprehensive analysis of constant-time polynomial inversion for post-quantum cryptosystems.". *In Proceedings of the 17th ACM International Conference on Computing Frontiers*, pages (pp. 269–276)., 2020, May.
4. Daniel J. Bernstein and Bo-Yin Yang. "Fast constant-time gcd computation and modular inversion.". *IACR transactions on cryptographic hardware and embedded systems*, pages 340–398, 2019.
5. T. Chang, E. Lu, Y. Lee, Y. Leu, and H. Shyu. "Two Algorithms for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Basis". *Information Processing Letters*, 1998.
6. Gong-Liang Chen, Yi-Yang Chen, Jian-Hua Li, and Yin Li. "An improvement of the TYT algorithm for $GF(2^m)$ based on reusing intermediate computation results". *Communications in Mathematical Sciences*, 9(1):277–287, 2011.
7. Nir Drucker, Shay Gueron, and Dusan Kostic. "Fast polynomial inversion for post quantum QC-MDPC cryptography". *Cyber Security Cryptography and Machine Learning: Fourth International Symposium, CSCML 2020, Be'er Sheva, Israel*, Cham: Springer International Publishing:340–398, July 2-3, 2020.
8. Andrea Galimberti, Gabriele Montanaro, and Davide Zoni. "Efficient and Scalable FPGA Design of $GF(2^m)$ Inversion for Post-Quantum Cryptosystems". *IEEE Transactions on Computers*, 71.12:3295–3307, 2022.
9. J. Hoffstein, D. Lieman, J. Pipher, and J. H. Silverman. "NTRU: A public key cryptosystem. NTRU Cryptosystems" . *Inc.(www.ntru.com)*, 1999.
10. Toshiya Itoh and Shigeo Tsujii. "A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases". *Information and Computation*, 78(3):171–177, 1988.
11. Walid Mustafa Mahmoud. *Speeding Up Finite Field Inversion for Cryptographic Applications*. University of Windsor (Canada), 2012.
12. J. Richter-Brockmann, M. S. Chen, S. Ghosh, and T. Güneysu. "Racing BIKE: Improved Polynomial Multiplication and Inversion in Hardware". *IACR Transactions on Cryptographic Hardware and Embedded Systems*, page 557–588, 2021.
13. J. Richter-Brockmann, J. Mono, and T. Güneysu. "Folding BIKE: Scalable Hardware Implementation for Reconfigurable Devices". *IEEE Transactions on Computers*, 71, no. 5: 1204–1215, 1 May 2022.
14. N. Takagi, J. I. Yoshiki, and K. Takagi. "A fast algorithm for multiplicative inversion in $GF(2^m)$ using normal basis". *IEEE Transactions on Computers*, 50(5): pp. 394–398, 2001.
15. D. Zoni, A. Galimberti, and W. Fornaciari. "Flexible and scalable FPGA-oriented design of multipliers for large binary polynomials". *IEEE Access*, 8,75809-75821, 2020.