Progetto di Ingegneria Informatica
Post-Quantum Cryptography

Domenico Cacace

Matricola 891291

Abstract

In questa relazione analizziamo l'algoritmo di inversione polinomiale proposto da D.J. Bernstein e B.Y. Yang, focalizzandoci in particolare su una sua implementazione per polinomi binari intesa per l'utilizzo all'interno di un crittosistema resistente ad attacchi di computer quantistici; passiamo quindi ad analizzare le criticità dell'implementazione, fornendo alcuni metodi per migliorare questi aspetti. Infine, valutiamo le performance e le proprietà dell'algoritmo comparandole con altri algoritmi per il medesimo scopo.

I. Introduzione

L' interesse per la ricerca nel campo della crittografia post-quantistica è cresciuto esponenzialmente con l'annuncio del National Institute of Standards and Technology dell'avvio del processo di valutazione e standardizzazione di uno o più algoritmi di crittografia asimmetrici resistenti ad attacchi di computer quantistici.[1]; tra i candidati troviamo LEDAcrypt[1, 2], basato sul crittosistema McEliece.
I crittosistemi basati sulla teoria dei codici, come appunto LEDAcrypt, hanno lo svantaggio di usare chiavi pubbliche piuttosto larghe; un modo per ridurre le dimensioni di queste chiavi è utilizzare famiglie di codici che consentono di essere rappresentati in modo più efficiente per quanto riguarda lo spazio occupato, come ad esempio i codici QC-MDPC (Quasi-cyclic moderate-density parity-check): questi codici sono composti da matrici a blocchi quadrate circolanti, dove ogni riga è ottenuta dallo shift circolare della precedente. L'aritmetica di queste matrici di dimensioni $p \times p$ è isomorfa a quella dei polinomi a coefficienti binari modulo $x^p - 1$, per cui possiamo sfruttare questa proprietà per ridurre il tempo di esecuzione del calcolo dell'inverso.

Organizzazione. Nella sezione II analizziamo in breve l'algoritmo di inversione e descriviamo come questo è stato *simulato*. Nella sezione III descriviamo come i polinomi sono memorizzati e come le relative operazioni aritmetiche sono implementate; passiamo quindi a descrivere le principali criticità dell'implementazione proposta in [3] e le relative ottimizzazioni sviluppate. Infine, nella sezione IV andiamo a comparare i risultati dei benchmark della nuova implementazione con gli algoritmi proposti in [4], con particolare attenzione ai tempi d'esecuzione e all'indipendenza di questi dagli input.

[1] https://csrc.nist.gov/projects/post-quantum-cryptography

II. Analisi dell'algoritmo

In [5] gli autori descrivono due metodi per il calcolo di massimo comun divisore e inverso moltiplicativo: il primo per numeri interi, il secondo per anelli di polinomi; nello specifico, andremo a sfruttare quest'ultimo per anelli di polinomi a coefficienti binari per le ragioni precedentemente citate.

L'algoritmo di inversione che andiamo ad analizzare è basato su un approccio *divide et impera*: il polinomio da invertire e il modulo vengono suddivisi dalla funzione `jumpdivstep`, che prende in input due polinomi $f(x), g(x)$, di grado al più $n = 2p - 1$, con $\delta = \deg(f(x)) - \deg(g(x))$ e individua un *pivot* per suddividere gli operandi in due metà[2], ovvero $j = \lfloor \frac{n}{2} \rfloor$.

La funzione `jumpdivstep` è quindi chiamata ricorsivamente passando come parametri le due metà dei polinomi appena individuate, finché la loro dimensione non è sufficientemente piccola (ovvero fin quando la dimensione non è pari o inferiore a quella di una *machine word*): a questo punto viene invocata la funzione `divstep`, che gestisce il caso base della ricorsione; i

[2] Questa scelta si è rivelata essere ottimale, come riportato in [4], ma qualsiasi valore di j compreso tra 1 ed $n - 1$ è valido

risultati trovati vengono quindi ricombinati fino ad ottenere il risultato finale.

Nello specifico, la funzione `invert` chiama la funzione `jumpdivstep` passando come parametri le rappresentazioni riflesse del modulo $f(x)$ e dell'elemento da invertire $a(x)$: per rappresentazione riflessa intendiamo il polinomio ottenuto invertendo il coefficiente del termine x^i con quello del termine $x^{\deg(S(x))-i}$, ottenuto tramite la funzione `mirror`; alla fine, la funzione ritorna la rappresentazione riflessa dell'inverso di $a(x)$, $H_{0,1}$, ottenuta dal prodotto dei risultati parziali calcolati dalle chiamate ricorsive.

Algorithm 1: `jumpdivstep`

Input: $f(x), g(x)$: polinomi binari
n: grado massimo di $f(x)$
$\delta = \deg f(x) - \deg g(x)$
Output: $P \times Q$: matrice di polinomi, risultato parziale dell'inversione
Data: ws: dimensione in bit di una *machine word*
P, Q: matrici di polinomi, contengono i risultati parziali delle due chiamate ricorsive

1 **function** `jumpdivstep`$(n, \delta, f(x), g(x))$**:**
2 **if** $n \leq ws$ **then**
3 **return** `divstep`$(n, \delta, f(x), g(x))$;
4 **end**
5 $j \leftarrow \lfloor \frac{n}{2} \rfloor$;
6 $\delta, P \leftarrow$ `jumpdivstep`$(j, \delta, f(x) \mod x^j, g(x) \mod x^j)$;
7 $f(\bar{x}) \leftarrow P_{0,0} \cdot f(x) + P_{0,1} \cdot g(x)$;
8 $g(\bar{x}) \leftarrow P_{1,0} \cdot f(x) + P_{1,1} \cdot g(x)$;
9 $\delta, Q \leftarrow$ `jumpdivstep`$(n - j, \delta, \frac{f(\bar{x})}{x^j}, \frac{g(\bar{x})}{x^j})$;
10 **return** $\delta, (P \times Q)$;

Algorithm 2: `invert`

Input: $f(x)$: polinomio irriducibile di $GF(2^p)$
$a(x)$: elemento invertibile di $GF(2^p)$
Output: $V(x)$, con $a^{-1}(x) \equiv V(x) \in GF(2^p)$

1 **function** `invert`$(f(x), a(x))$**:**
2 $S(x) \leftarrow$ `mirror`$(f(x))$;
3 $R(x) \leftarrow$ `mirror`$(a(x))$;
4 $\delta, H \leftarrow$ `jumpdivstep`$(2p - 1, 1, S(x), R(x))$;
5 $V(x) \leftarrow$ `mirror`$(H_{0,1})$;
6 **return** $V(x)$;

Simulazione dell'algoritmo

Possiamo, partendo dall'algoritmo qui presentato, ricostruire un albero di ricorsione binario tracciando tutte le chiamate ed i vari parametri necessari: la prima chiamata a `jumpdivstep` (`invert`, riga 4) è la radice dell'albero, la prima chiamata effettuata (`jumpdivstep`, riga 6) corrisponde al figlio destro di un nodo, mentre la seconda (`jumpdivstep`, riga 9) corrisponde al figlio sinistro; quando la dimensione degli operandi è pari o inferiore alla *word size*, ossia quando viene invocata la funzione `divstep`, il nodo corrispondente nell'albero di ricorsione risulta essere una foglia.

Analizzando l'albero di ricorsione ed i parametri relativi ai nodi, possiamo inoltre determinare le dimensioni dei vettori che contengono i risultati intermedi (P, Q) e gli operandi intermedi $(f(\bar{x}), g(\bar{x}))$, nonché gli indici di accesso ai suddetti per ogni chiamata. Da qui, possiamo *simulare* le chiamate attraversando l'albero usando un approccio *depth-first*, dando priorità ai figli destri (che per costruzione corrispondono alla prima chiamata). È inoltre interessante notare che le operazioni ed i parametri necessari per ogni chiamata dipendano solo dalla dimensione dei parametri di input, e non dai parametri stessi: dato che le dimensioni sono note tramite un'analisi statica, possiamo derivare l'albero di ricorsione a compile time; questo ci permetterà di effettuare tutte le ottimizzazioni riportate in III, e quindi poter generare una versione iterativa di `jumpdivstep` specifica per dimensione dell'input.

Contenuto dei nodi

Ad ogni nodo dell'albero di ricorsione corrisponde una chiamata a `jumpdivstep`, ognuna caratterizzata dai propri parametri, tra cui:

- `n`: grado massimo dei operandi ricevuti in input
- `j`: grado massimo degli operandi passati alla prima chiamata ricorsiva (figlio destro)
- `n-j`: grado massimo degli operandi passati alla seconda chiamata ricorsiva (figlio sinistro)
- `num_digits_(n|j|nminusj)`: dimensione degli operandi di cui sopra, espressa in `digit`

III. Implementazione ed Ottimizzazione

Rappresentazione dei polinomi

I polinomi, elementi dell'anello $GF(2)[x]$, sono salvati in forma binaria compatta: ogni bit in un byte rappresenta

Table 1: *Rappresentazione dei polinomi, singolo byte*

Bin	Hex	Polinomio
0000 0000	0x00	0
0000 0001	0x01	1
0000 0002	0x02	x
0000 0003	0x03	$x+1$
...	...	...
0000 1111	0x0F	x^3+x^2+x+1
...	...	...
1111 1111	0xFF	$x^7+x^6+x^5+\cdots+x+1$

un coefficiente di un polinomio binario in formato Big-Endian (leggendo da sinistra a destra, i primi elementi che si incontrano sono i coefficienti di grado più alto)

Una *machine word* è considerata un `digit` ed i byte che compongono un `digit` sono in formato Big Endian; allo stesso modo, anche gli elementi con più `digit` sono rappresentati in formato Big Endian.

Estensioni dell'ISA x86

L'implementazione proposta in [3] utilizza AVX, un'estensione dell' Instruction Set Architecture di x86 che permette di lavorare su più parole di memoria in parallelo, portando notevoli vantaggi nelle operazioni aritmetiche: ciò significa, su una macchina a 64 bit (dimensione di un `digit`), lavorare su 128 o 256 bit. Questo ci permette inoltre di avere un'implementazione molto efficiente per i prodotti, utilizzando funzioni altamente specializzate per polinomi di 8 `digit` o meno e loro combinazioni per dimensioni maggiori, come riportato in [6]; gli intrinsics, funzioni gestite direttamente dal compilatore che permettono di usare istruzioni specifiche dell'architettura per cui si sta compilando come fossero chiamate a funzioni standard, permettono inoltre l'implementazione delle diverse versioni di `divstep`.

Punti critici

Tramite un'analisi di complessità asintotica, supportata dai risultati sperimentali ottenuti tramite `callgrind`, notiamo che la maggior parte del tempo di esecuzione è dovuta a `gf2x_scalarprod`: ciò non ci sorprende, in quanto questa funzione è invocata due volte per il calcolo di $f(\bar{x})$ e $g(\bar{x})$ (`jumpdivstep`, righe 7-8), e altre quattro per $P \times Q$ (`jumpdivstep`, riga 10); inoltre, per quest'ultima operazione, sono necessarie altrettante chiamate a `memcpy`, per spostare i risultati ottenuti da un vettore temporaneo alla destinazione desiderata.

Algorithm 3: `gf2x_scalarprod` (per $na > nb$)

Input: a_0, a_1, b_0, b_1: array di polinomi
na, nb: dimensioni degli array di polinomi
Output: $res = a_0 \cdot b_0 + a_1 \cdot b_1$, dimensione $na + nb$

```
1  function scalarprod(na, a0, a1, nb, b0, b1):
2      DIGIT tmp[na * 2];
3      DIGIT tmp2[na * 2];
4      DIGIT buffer[na];
5      memset(buffer, 0x00, (na − nb)*DIGIT_SIZE_B);
6      memcpy(buffer+(na − nb), b0, nb*DIGIT_SIZE_B);
7      GF2X_MUL(na * 2, tmp, a0, buffer);
8      memcpy(buffer+(na − nb), b1, nb*DIGIT_SIZE_B);
9      GF2X_MUL(na * 2, tmp2, a1, buffer);
10     gf2x_add(na * 2, tmp, tmp, tmp2);
11     memcpy(res, tmp+(na − nb),
        (na + nb)*DIGIT_SIZE_B);
12     return res
```

La funzione `gf2x_scalarprod` prende in input due coppie di polinomi $(a_0, a_1,\ b_0, b_1)$, di dimensione rispettivamente na e nb `digit`, e produce in output il prodotto scalare $res = a_0 \cdot b_0 + a_1 \cdot b_1$, di dimensione $nr = na + nb$; in realtà, per poter eseguire le moltiplicazioni tra polinomi, è necessario spostare l'operando più corto in un buffer (righe 6, 8) e aggiungere zeri nelle posizioni dei coefficienti più significativi mancanti (riga 5), fino a raggiungere la dimensione dell'altro operando.

Una volta calcolati i prodotti, questi vengono ulteriormente manipolati, troncando i coefficienti superflui, shiftandoli (`jumpdivstep`, righe 7-8) e spostandoli da una posizione all'altra; analizzando come questi polinomi vengono manipolati, possiamo andare a salvare i risultati direttamente nelle posizioni corrette, riduendo i tempi di esecuzione.

Ottimizzazioni

Allocazione dei polinomi

Il primo passo nell'unrolling dell'algoritmo ricorsivo consiste nell'allocare in una volta sola tutta la memoria necessaria per salvare i risultati e gli operandi intermedi, ossia $P, Q, f(\bar{x})$ e $g(\bar{x})$; le stesse porzioni di memoria vengono riutilizzate da diverse chiamate in momenti diversi, per cui viene presa la dimensione massima per ogni *slot*, individuabile percorrendo il ramo destro dell'albero di ricorsione dalla radice all'ultima foglia. Partendo con operandi di grado $n_0 - 1$, `digit` di

dimensione s e parole di memoria di dimensione ws (entrambe espresse in bit), avremo un albero di ricorsione di profondità $d = \lceil \log_2 \frac{n}{s} \rceil$; per ogni livello, avremo

$$j_i = \left\lfloor \left(\left\lfloor \frac{n_i}{2} \right\rfloor + ws - 2 \right) \cdot \frac{1}{ws-1} \right\rfloor \cdot (ws-1)$$

$$n_i = j_{i-1} \qquad (i \neq 0)$$

Le operazioni di somma, prodotto e divisione che includono ws sono necessarie ad ottenere, alla fine della ricorsione, polinomi di grado $ws - 1$[3], in modo da sfruttare appieno gli intrinsics di AVX.

Partendo da quanto appena trovato, possiamo calcolare il numero di `digit` necessario a memorizzare i vari polinomi:

$$\texttt{num\_digits\_n}_\text{i} = \left\lceil \frac{n_i}{s} \right\rceil$$

$$\texttt{num\_digits\_j}_\text{i} = \left\lceil \frac{j_i}{s} \right\rceil$$

$$\texttt{num\_digits\_nminusj}_\text{i} = \left\lceil \frac{n_i - j_i}{s} \right\rceil$$

Possiamo quindi individuare le dimensioni degli slot dei vari array in base al livello di profondità i:

- P: d slot di dimensione $\texttt{num\_digits\_j}_i$, 4 array
- P: d slot di dimensione $\texttt{num\_digits\_nminusj}_i$, 4 array
- $f(\bar{x})$: $d-1$ slot di dimensione $\texttt{num\_digits\_n}_i + \texttt{num\_digits\_j}_i$, 1 array
- $g(\bar{x})$: come al punto precedente.

Tenendo traccia delle dimensioni dei vari componenti è poi possibile determinare gli indici di accesso agli array per i vari valori di d.

Scomposizione dei prodotti

Come accenato, le funzioni per il calcolo del prodotto tra polinomi richiedono operandi di dimensione uguale: invece di aggiungere coefficienti nulli al polinomio più corto ($a = 0, a_1$), possiamo dividere il polinomio più lungo ($b = b_0, b_1$) in due parti:

$$\begin{array}{cccc} 0 & a_1 & \times \\ b_0 & b_1 & = \\ \hline 0 \cdot b_1 & a_1 \cdot b_1 & + \\ 0 \cdot b_0 & a_1 \cdot b_0 & - \end{array}$$

[3]In alcuni casi i polinomi ottenuti sono di grado inferiore per *mancanza di coefficienti*

la parte *inferiore* (b_1, coefficienti meno significativi), della stessa dimensione di a_1, viene moltiplicata direttamente con a_1 stesso; il procedimento viene quindi ripetuto considerando ora il polinomio a_1 e la parte restante dell'altro polinomio, b_0, fino ad esaurimento.

I vari sottoprodotti che vengono calcolati vanno quindi sommati: dati due sottoprodotti successivi r_i, r_{i+1}, abbiamo che $\deg(r_{i+1}) - \deg(r_i) = na$ (in modo simile a quanto sopra, na indica la dimensione del poliniomio più corto alla i-esima iterazione): con riferimento all'esempio, abbiamo che $r_1 = a_1 \cdot b_1$ e $r_2 = a_1 \cdot b_0$, per cui i na coefficienti meno significativi di r_1 vengono sommati ai na coefficienti più significativi di r_2, mentre i restanti (na per r_1, nb per r_2) vengono semplicemente riportati come sono nel risultato, in quanto i sottoprodotti restanti sono nulli.

Quindi, nell'implementazione del generatore, sostituiamo ogni chiamata a `gf2x_scalarprod` con la corrispondente sequenza di `gf2x_add` e `GF2X_MUL`, utilizzando un accumulatore, nr, per tenere traccia dell'offset da considerare; inoltre, nel caso i due operandi siano di dimensioni inferiori agli 8 `digit`, andiamo ad utilizzare direttamente le funzioni specifiche per la dimensione desiderata (`gf2x_mul_X_avx`), evitando le chiamate a `GF2X_MUL`.

Troncamento e shift dei risultati

In quasi tutti i casi l'output di `gf2x_scalarprod` viene troncato, eliminando sempre una porzione dei coefficienti più significativi che eccedono la dimensione necessaria; dato che il metodo di moltiplicazione parte dai coefficienti meno significativi, possiamo bloccarne l'esecuzione prematuramente appena abbiamo tutti i coefficienti necessari, evitando il calcolo di quelli che sarebbero poi tagliati. Ciò viene banalmente implementato nel generatore utilizzando lo stesso accumulatore nr citato al punto prima, ma sottraendovi il numero di `digit` che sarebbero eliminati successivamente.

A questo punto, possiamo quindi evitare di utilizzare un vettore ausiliario per il calcolo dei prodotti (`jumpdivstep`, righe 6-8, 10), ma ne abbiamo ancora bisogno per il calcolo di $\frac{f(\bar{x})}{x^j}$ e $\frac{g(\bar{x})}{x^j}$: queste divisioni sono infatti ottenute tramite uno shift a sinistra di j bit; per risolvere questo problema, dividiamo lo shift in due parti:

- shift di $(\lfloor \frac{j}{\texttt{DIGIT\_SIZE\_b}}$[4]$\rfloor)$ `digit` interi;
- shift di ($j \mod$ `DIGIT_SIZE_b`) bit singoli.

[4]Dimensione di un `digit` in bit

Lo shift dei `digit` interi può essere ricondotto al problema del troncamento dei coefficienti più significativi, aggiustando opportunamente gli offset; per quanto riguarda lo shift dei singoli bit (che saranno per costruzione meno di quelli in un `digit`), abbiamo bisogno di aggiungere un `digit` in più per ognuno dei polinomi $f(\bar{x})$ e $g(\bar{x})$, in modo da poter shiftare i bit del polinomio desiderato senza andare a sovrascrivere il `digit` meno significativo del polinomio precedente.

IV. Risultati

Il framework [3], oltre a contenere l'implementazione di alcuni algoritmi di inversione polinomiale (tra cui la versione di partenza dell'algoritmo di Bernstein-Yang), fornisce dei tool per testare la performance dei vari algoritmi per poi compararli; sono stati effettuati diversi test durante lo sviluppo, ma per brevità riportiamo solo i risultati dell'implementazione originale e quella finale. Ogni test prevede l'esecuzione di 1000 round per ogni p di interesse; in ogni round viene misurato il tempo di esecuzione dell'algoritmo, prima con un input casuale, poi con un trinomio $(x^2 + x + 1)$:[5] questo viene fatto per calcolare la t di Student per le due popolazioni e verificare che il tempo di esecuzione degli algoritmi sia costante in funzione dell'input (non della sua dimensione), garantendo che il crittosistema che lo usa sia immune (in questo punto) ad attacchi *side channel* che puntano a risalire all'input analizzando i tempi di esecuzione.

I test sono stati effettuati su una macchina AMD64, con processore AMD Ryzen 5 2500U (disabilitando il TurboBoost e bloccando la frequenza a 1.9GHz per ottenere risultati consistenti); l'implementazione è stata compilata con GCC 10.2.0, abilitando la generazione di codice specifico per l'architettura utilizzata (`-march=native`) e abilitando il livello massimo di ottimizzazione (`-O3`).

Analizzando i risultati, notiamo che la *nuova* versione dell'algoritmo di Bernstein-Yang ha un miglioramento dal punto di vista temporale quasi costante per tutti i valori di p (+18.56%), rimanendo più lento dell'algoritmo KTT [7] per $p < 35507$ (rispetto a $p < 48371$ della versione originale); d'altro canto, i valori della t di Student per BY sono inferiori in modulo a 4.5, la soglia sotto cui possiamo affermare che il tempo di esecuzione dell'algoritmo è indipendente dall'input con un livello di confidenza del 99.999%, a differenza di KTT.

[5] In ogni caso, l'inversione viene calcolata prendendo come modulo $x^p - 1$

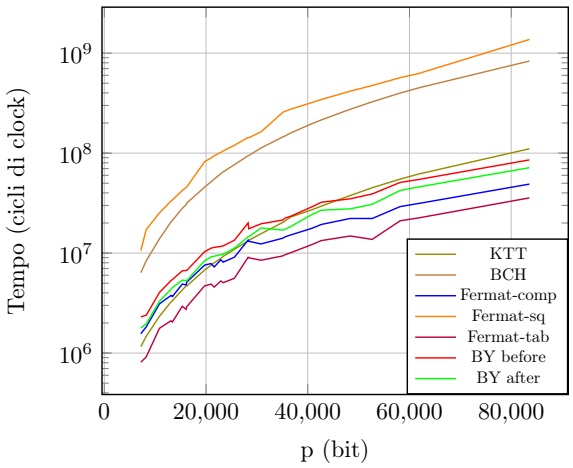

Figure 1: *Tempi di esecuzione*

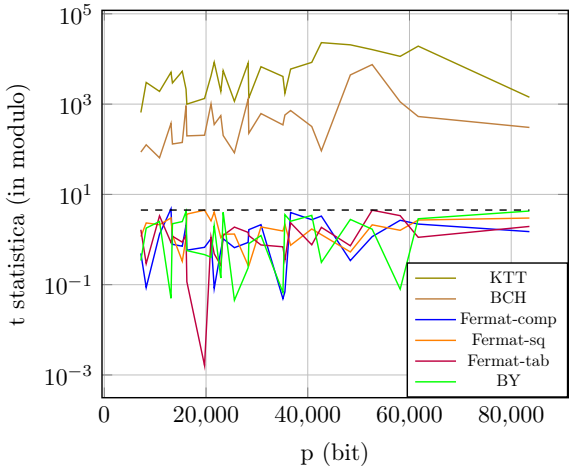

Figure 2: *Valori della t di Student*

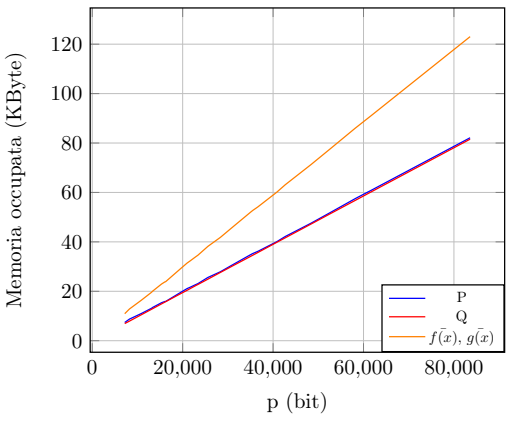

Figure 3: *Memoria occupata*

References

[1] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, "LEDAcrypt: QC-LDPC code-based cryptosystems with bounded decryption failure rate," in *Code-Based Cryptography Workshop*, pp. 11–43, Springer, 2019.

[2] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, "LEDAkem: a post-quantum key encapsulation mechanism based on QC-LDPC codes," 01 2018.

[3] A. Barenghi and G. Pelosi, "A self-contained software library to benchmark binary polynomial multiplicative inverses," Apr. 2020.

[4] A. Barenghi and G. Pelosi, "A comprehensive analysis of constant-time polynomial inversion for post-quantum cryptosystems," pp. 269–276, 05 2020.

[5] D. J. Bernstein and B.-Y. Yang, "Fast constant-time gcd computation and modular inversion," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 340–398, 2019.

[6] M. Bodrato, "Towards optimal toom-cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0," in *International Workshop on the Arithmetic of Finite Fields*, pp. 116–133, Springer, 2007.

[7] N. Takagi, J.-i. Yoshiki, and K. Takagi, "A fast algorithm for multiplicative inversion in gf (2/sup m/) using normal basis," *IEEE Transactions on Computers*, vol. 50, no. 5, pp. 394–398, 2001.