Progetto di Ingegneria Informatica
Post-Quantum Cryptography

Domenico Cacace

Matricola 891291

Abstract

In questa relazione analizziamo l'implementazione dell'algoritmo di inversione polinomiale presentato da Bernstein e Yang [1], analizzandone le caratteristiche e comparando le misurazioni dei tempi di esecuzione tramite un apposito framework [2, 3]. Da qui, provvederemo a creare un generatore di codice sorgente per ottimizzare l'implementazione delle operazioni più frequenti nell'algoritmo e ridurne il tempo di esecuzione.

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[4, 5], basato sul crittosistema McEliece.
I crittosistemi basati sulla teoria dei codici 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 modulo $x^p - 1$, per cui possiamo sfruttare questa proprietà per ridurre il tempo di esecuzione del calcolo dell'inverso.

II. Analisi dell'algoritmo

L'algoritmo proposto in [1] è basato su un approccio *divide et impera*: l'operando viene diviso dalla funzione `jumpdivstep`, che prende in input due polinomi f, g, di grado al più n e con $\delta = \deg f - \deg g$ e individua un *pivot* j, che divide i polinomi a metà. 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 (vedi III); questo punto viene invocata la funzione `divstep`, che gestisce il caso base della ricorsione. La funzione `jumpdivstep` prende come input di partenza la rappresentazione riflessa di modulo $f(x)$ ed elemento da invertire $a(x)$, e ritorna alla fine la rappresentazione riflessa dell'inverso di $a(x)$, $\mathcal{H}_{0,1}$, con $V(x) \equiv a^{-1}(x)$.

Algorithm 1: jumpdivstep

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 $\delta, \mathcal{P} \leftarrow$ `jumpdivstep`$(j, \delta, f(x) \mod x^j, g(x) \mod x^j)$;
7 $f'(x) \leftarrow \mathcal{P}_{0,0} \cdot f(x) + \mathcal{P}_{0,1} \cdot g(x)$;
8 $g'(x) \leftarrow \mathcal{P}_{1,0} \cdot f(x) + \mathcal{P}_{1,1} \cdot g(x)$;
9 $\delta, \mathcal{Q} \leftarrow$ `jumpdivstep`$(n - j, \delta, \frac{f'(x)}{x^j}, \frac{g'(x)}{x^j})$;
10 **return** $\delta, (\mathcal{P} \times \mathcal{Q})$
11 **function** `invert`$(f(x), a(x))$**:**
12 $S(x) \leftarrow$ `MIRROR`$(f(x))$;
13 $R(x) \leftarrow$ `MIRROR`$(a(x))$;
14 $\delta, \mathcal{H} \leftarrow$ `jumpdivstep`$(2m - 1, 1, S(x), R(x))$;
15 $V(x) \leftarrow$ `MIRROR`$(\mathcal{H}_{0,1})$;
16 **return** $V(x)$;

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

Simulazione dell'algoritmo

Possiamo, partendo dall'algoritmo qui presentato, ricostruire un albero di ricorsione tracciando tutte le chiamate ed i vari parametri necessari: la prima chiamata a `jumpdivstep` (riga 14) è la radice dell'albero, la prima chiamata effettuata (riga 6) corrisponde al figlio destro di un nodo, mentre la seconda (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 $(\mathcal{P}, \mathcal{Q})$ e gli operandi intermedi $(f'(x), g'(x))$, 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; 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)`: numero di `DIGIT` (vedi III) necessari a contenere gli operandi di cui sopra

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 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)

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	$x^7+x^6+x^5+\cdots+x+1$

Table 1: *Rappresentazione dei polinomi, singolo byte*

Una *machine word* A_i 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 [2] 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 [6]; gli intrinsics permettono anche l'implementazione delle diverse versioni di `divstep`.

Punti critici

Analizzando l'esecuzione dell'algoritmo 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 $f'(x)$ e $g'(x)$ (righe 7-8), e altre quattro per $\mathcal{P} \times \mathcal{Q}$ (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.

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 e aggiungere zeri nelle posizioni dei coefficienti più significativi mancanti, fino a raggiungere la dimensione dell'altro operando.

Una volta calcolati i prodotti, questi vengono ulteriormente manipolati, troncando i coefficienti superflui, shiftandoli (calcolo di f' e g', 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 $\mathcal{P}$, $\mathcal{Q}$, f' e g'; 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. Abbiamo quindi, con un albero di ricorsione di profondità d:

- $\mathcal{P}$: 4 array composti da d componenti, ognuno di dimensione `num_digits_j`;
- $\mathcal{Q}$: 4 array composti da d componenti, ognuno di dimensione `num_digits_nminusj`;
- f': 1 array composto da $d-1$ componenti, ognuno di dimensione `num_digits_n` + `num_digits_j`;
- g': 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), possiamo dividere il polinomio più lungo (b) in due parti: la parte *inferiore* (b_0, coefficienti meno significativi), della stessa dimensione di a, viene moltiplicata direttamente con a stesso; il procedimento viene quindi ripetuto considerando ora il polinomio a e la parte restante dell'altro polinomio, b_1, 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).

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; 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 (righe 6-8, 10), ma ne abbiamo ancora bisogno per il calcolo di $(f' \div x^j)$ e $(g' \div x^j)$: 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 $(\lfloor j \div \texttt{DIGIT\_SIZE\_b}^2 \rfloor)$ `DIGIT` interi;
- shift di ($j \mod$ `DIGIT_SIZE_b`) bit singoli.

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' e g', 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 [2], oltre a contenere l'implementazione di alcuni algoritmi di inversione polinomiale (tra cui la

[2] Dimensione di un `DIGIT` in bit

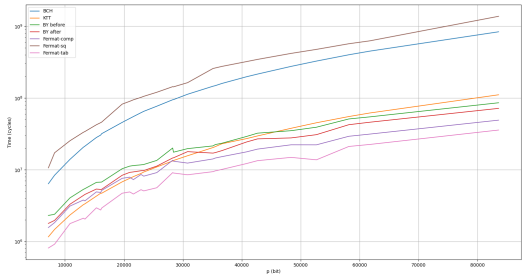

Figure 1: *Tempi di esecuzione*

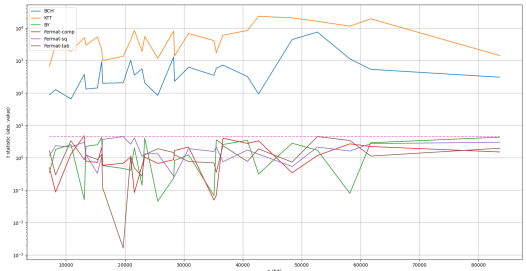

Figure 2: *Valori della t statistica*

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 $(x^2 + x + 1)$:[3] 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.

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

References

[1] 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.

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

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

[4] 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.

[5] 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.

[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.