A note on the implementation of the Number Theoretic Transform

Mike Scott

MIRACL.com

16th IMA International Conference on Cryptography and Coding, December 2017

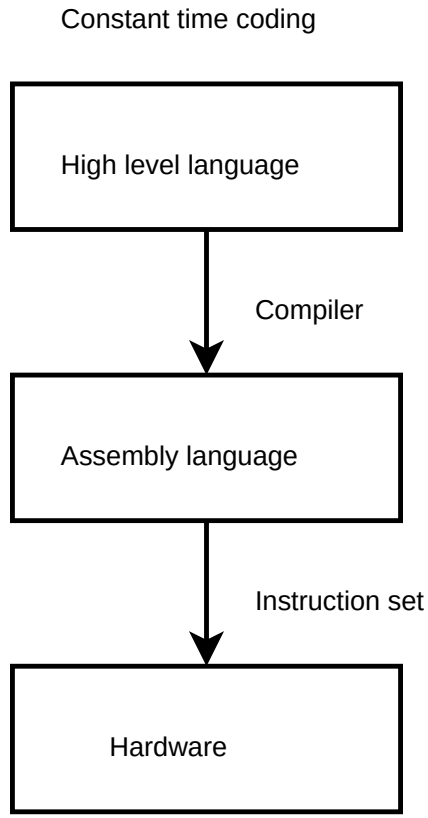

Figure: Constant Time Programming

Constant Time Programming

... is good for crypto!

- Use exception-free algorithms
- Program without **if** statements!
- Program with fixed length loops
- Results in straight-line code
- Assuming compiler doesn't mess things up
- Assuming it generates only constant time (data independent) instructions
- Bonus – just one path through the code – easy to debug!

Ring Learning With Errors

- Looks like a potential winner in the Post-Quantum stakes
- Order of magnitude faster than ECC ☺
- Order of magnitude larger keys ☹
- Quantum hard one-way function $P = As + e$
- Which can be trap-doored.
- Mostly easy to program in constant time (though statistical sampling is difficult)

The Number Theory Transform

- Very fast FFT based method for polynomial multiplication
- Works best for polynomials with degree $n = 2^m$
- Polynomial arithmetic modulo irreducible cyclotomic $X^n + 1$
- and prime $q = 1 \bmod 2n$, for roots of unity
- A lot of prior art (e.g. Longa & Naehrig, CANS 2016)

Cooley-Tukey NTT

Algorithm 1 The Cooley-Tukey NTT algorithm

INPUT: A vector $\vec{x} = [x_0, \ldots, x_{n-1}]$ where $x_i \in [0, p-1]$ of degree n (a power of 2) and modulus $q = 1 \bmod 2n$
INPUT: Precomputed table of $2n$-th roots of unity $\vec{g}$, in bit reversed order
OUTPUT: $\vec{x} \leftarrow NTT(\vec{x})$

1: **function** NTT(x)
2: $t \leftarrow n/2$
3: $m \leftarrow 1$
4: **while** $m < n$ **do**
5: $k \leftarrow 0$
6: **for** $i \leftarrow 0;\ i < m;\ i \leftarrow i + 1$ **do**
7: $S \leftarrow g[m + i]$
8: **for** $j \leftarrow k;\ j < k + t;\ j \leftarrow j + 1$ **do**
9: $U \leftarrow x[j]$
10: $V \leftarrow x[j + t].S \bmod q$
11: $x[j] \leftarrow U + V \bmod q$
12: $x[j + t] \leftarrow U - V \bmod q$
13: $k \leftarrow k + 2t$
14: $t \leftarrow t/2$
15: $m \leftarrow 2m$
16: **return**

The Gentleman-Sande INTT

Algorithm 2 The Gentleman-Sande inverse NTT algorithm

INPUT: A vector $\vec{x} = [x_0, \ldots, x_{n-1}]$ where $x_i \in [0, p-1]$ of degree n (a power of 2) and modulus $q = 1 \bmod 2n$

INPUT: Precomputed table of inverses of $2n$-th roots of unity $\vec{g^{-1}}$, in bit reversed order

INPUT: $n^{-1} \bmod q$

OUTPUT: $\vec{x} \leftarrow INTT(\vec{x})$

1: **function** INTT(x)
2: $t \leftarrow 1$
3: $m \leftarrow n/2$
4: **while** $m > 0$ **do**
5: $k \leftarrow 0$
6: **for** $i \leftarrow 0;\ i < m;\ i \leftarrow i + 1$ **do**
7: $S \leftarrow g^{-1}[m + i]$
8: **for** $j \leftarrow k;\ j < k + t;\ j \leftarrow j + 1$ **do**
9: $U \leftarrow x[j]$
10: $V \leftarrow x[j + t]$
11: $x[j] \leftarrow U + V \bmod q$
12: $W \leftarrow U - V \bmod q$
13: $x[j + t] \leftarrow W.S \bmod q$
14: $k \leftarrow k + 2t$
15: $t \leftarrow 2t$
16: $m \leftarrow m/2$
17: **for** $i \leftarrow 0;\ i < n;\ i \leftarrow i + 1$ **do**
18: $x[i] \leftarrow x[i].n^{-1} \bmod q$
19: **return**

Montgomery Modular Reduction

- Gives us modular reduction without division
- Choose a new modulus $R > q$. In most cases $R = 2^{32}$, is a good choice.
- Field elements are converted to, and processed in n-residue form
- Reduction function **redc** uses only multiplication (as reduction modulo R is easy).
- For $Z < qR$, then $\text{redc}(Z) = Z \bmod 2q$

Lazy reduction

- Delay modular reduction as long as possible, ideally until the end of the calculation
- Consider a calculation involving a sequence of modular multiplications, additions and subtractions.
- ...as arises in the NTT (and elliptic curve point doubling/addition)
- Montgomery reduction naturally helps dampens excesses that arise! Barret reduction doesn't do that...

A simple example

- Assume $R > 64q$ and consider the modular squaring of a value $x < 8q$. So $x^2 < 64q^2 < qR$.
- It can be assumed that x has increased well above q due to a sequence of lazy additions, without reduction.
- But after squaring followed by Montgomery reduction, then $x < 2q$!
- Indeed any excessive value can be (almost fully) reduced by simply performing a Montgomery modular multiplication by the n-residue equivalent of unity.

Full reduction

- Eventually residues might need to be fully reduced from $< 2q$ to $< q$
- This can be achieved by a simple constant-time trick
- Assume the residue x is stored as a 32-bit signed integer.

```
x=x-q;
x+=(x>>31)&q;
```

Slothful Reduction – a simple idea

- Temporarily add code to monitor worst case excesses. Precomputed data has an excess of $e = 1$, meaning $x < q$
- Outputs of Montgomery reduction have an excess of $e = 2$, so $x < 2q$.
- Modular negation is $-x = eq - x$, where e is the current excess of x.
- Observe worst-case excesses, and check for overflow.
- Print out and examine logs (Remember its straight-line code, so excesses are invariant)
- Remove extra code. Replace modular negation with $-x = Eq - x$, where E is worst-case.

Excess arithmetic

- Modular addition or subtraction increases excesses
- So if $c = a + b$, the $e_c = e_a + e + b$
- Modular multiplication reduces excesses
- Behaviour is reminiscent of a dynamic system
- Behaviour can be stable, or unstable. Depends on the algorithm.

Excess analysis for NTT

- the “butterfly” pattern is quite hard to follow..!
- NTT and Inverse NTT algorithms behave quite differently!
- As the iterations progress, the excesses get bigger
- For Cooley-Tukey NTT they increase linearly – quasi-stable
- For Gentlemen-Sande they increase exponentially – unstable!

Cooley-Tukey NTT – Before

Algorithm 3 The Cooley-Tukey NTT algorithm

INPUT: A vector $\vec{x} = [x_0, \ldots, x_{n-1}]$ where $x_i \in [0, p-1]$ of degree n (a power of 2) and modulus $q = 1 \bmod 2n$
INPUT: Precomputed table of $2n$-th roots of unity $\vec{g}$, in bit reversed order
OUTPUT: $\vec{x} \leftarrow NTT(\vec{x})$

1: **function** NTT(x)
2: $t \leftarrow n/2$
3: $m \leftarrow 1$
4: **while** $m < n$ **do**
5: $k \leftarrow 0$
6: **for** $i \leftarrow 0;\ i < m;\ i \leftarrow i + 1$ **do**
7: $S \leftarrow g[m + i]$
8: **for** $j \leftarrow k;\ j < k + t;\ j \leftarrow j + 1$ **do**
9: $U \leftarrow x[j]$
10: $V \leftarrow x[j + t].S \bmod q$
11: $x[j] \leftarrow U + V \bmod q$
12: $x[j + t] \leftarrow U - V \bmod q$
13: $k \leftarrow k + 2t$
14: $t \leftarrow t/2$
15: $m \leftarrow 2m$
16: **return**

Cooley-Tukey NTT – After

Algorithm 4 The Cooley-Tukey NTT algorithm

INPUT: A vector $\vec{x} = [x_0, \ldots, x_{n-1}]$ where $x_i \in [0, p-1]$ of degree n (a power of 2) and modulus $q = 1 \bmod 2n$
INPUT: Precomputed table of $2n$-th roots of unity $\vec{g}$, in bit reversed order
OUTPUT: $\vec{x} \leftarrow NTT(\vec{x})$

```
1:  function NTT(x)
2:      t ← n/2
3:      m ← 1
4:      while m < n do
5:          k ← 0
6:          for i ← 0; i < m; i ← i + 1 do
7:              S ← g[m + i]
8:              for j ← k; j < k + t; j ← j + 1 do
9:                  U ← x[j]
10:                 V ← x[j + t].S
11:                 x[j] ← U + V
12:                 x[j + t] ← U + 2q − V
13:             k ← k + 2t
14:         t ← t/2
15:         m ← 2m
16:     return
```

The Gentleman-Sande INTT – Before

Algorithm 5 The Gentleman-Sande inverse NTT algorithm

INPUT: A vector $\vec{x} = [x_0, \ldots, x_{n-1}]$ where $x_i \in [0, p-1]$ of degree n (a power of 2) and modulus $q = 1 \bmod 2n$

INPUT: Precomputed table of inverses of $2n$-th roots of unity $\vec{g^{-1}}$, in bit reversed order

INPUT: $n^{-1} \bmod q$

OUTPUT: $\vec{x} \leftarrow INTT(\vec{x})$

1: **function** INTT(x)
2: $t \leftarrow 1$
3: $m \leftarrow n/2$
4: **while** $m > 0$ **do**
5: $k \leftarrow 0$
6: **for** $i \leftarrow 0;\ i < m;\ i \leftarrow i + 1$ **do**
7: $S \leftarrow g^{-1}[m + i]$
8: **for** $j \leftarrow k;\ j < k + t;\ j \leftarrow j + 1$ **do**
9: $U \leftarrow x[j]$
10: $V \leftarrow x[j + t]$
11: $x[j] \leftarrow U + V \bmod q$
12: $W \leftarrow U - V \bmod q$
13: $x[j + t] \leftarrow W.S \bmod q$
14: $k \leftarrow k + 2t$
15: $t \leftarrow 2t$
16: $m \leftarrow m/2$
17: **for** $i \leftarrow 0;\ i < n;\ i \leftarrow i + 1$ **do**
18: $x[i] \leftarrow x[i].n^{-1} \bmod q$
19: **return**

The Gentleman-Sande INTT – After

Algorithm 6 The Gentleman-Sande inverse NTT algorithm

INPUT: A vector $\vec{x} = [x_0, \ldots, x_{n-1}]$ where $x_i \in [0, p-1]$ of degree n (a power of 2) and modulus $q = 1 \bmod 2n$

INPUT: Precomputed table of inverses of $2n$-th roots of unity $\vec{g^{-1}}$, in bit reversed order

INPUT: $n^{-1} \bmod q$

OUTPUT: $\vec{x} \leftarrow INTT(\vec{x})$

1: **function** INTT(x)
2: $t \leftarrow 1$
3: $m \leftarrow n/2$
4: **while** $m > 0$ **do**
5: $k \leftarrow 0$
6: **for** $i \leftarrow 0;\ i < m;\ i \leftarrow i + 1$ **do**
7: $S \leftarrow g^{-1}[m + i]$
8: **for** $j \leftarrow k;\ j < k + t;\ j \leftarrow j + 1$ **do**
9: $U \leftarrow x[j]$
10: $V \leftarrow x[j + t]$
11: $x[j] \leftarrow U + V$
12: $W \leftarrow U + nq - V$
13: $x[j + t] \leftarrow W.S$
14: $k \leftarrow k + 2t$
15: $t \leftarrow 2t$
16: $m \leftarrow m/2$
17: **for** $i \leftarrow 0;\ i < n;\ i \leftarrow i + 1$ **do**
18: $x[i] \leftarrow x[i].n^{-1} \bmod q$
19: **return**

Avoiding overflow for larger q

- This works fine for small q, for example $q = 12289$ (14 bits) on a 32-bit processor
- But some protocols need larger q to handle noise build-up
- In this case we would have a problem for the Inverse NTT, as nq might overflow.
- So go back to excess transcripts, and figure out where its necessary to force a reduction
- Experimentally we came up with the following solution. The performance impact is minimal

The Gentleman-Sande INTT

Algorithm 7 The Gentleman-Sande inverse NTT algorithm

1: **function** INTT(x)
2: $t \leftarrow 1$
3: $m \leftarrow n/2$
4: **while** $m > 0$ **do**
5: $k \leftarrow 0$
6: **for** $i \leftarrow 0;\ i < m;\ i \leftarrow i + 1$ **do**
7: $S \leftarrow g^{-1}[m + i]$
8: **for** $j \leftarrow k;\ j < k + t;\ j \leftarrow j + 1$ **do**
9: **if** $m < L \wedge j < k + L/(2m)$ **then**
10: $U \leftarrow x[j].O$
11: $V \leftarrow x[j + t].O$
12: **else**
13: $U \leftarrow x[j]$
14: $V \leftarrow x[j + t]$
15: $x[j] \leftarrow U + V$
16: $W \leftarrow U + (n/L)q - V$
17: $x[j + t] \leftarrow W.S$
18: $k \leftarrow k + 2t$
19: $t \leftarrow 2t$
20: $m \leftarrow m/2$
21: **return**

- L is chosen as smallest power of 2 such that $2(n/L)q < 2^{31}$
- and O is unity in Montgomery form
- Loop unrolling gets rid of the **if** statement
- Performance impact is minimal

Wrap-up

- A simple method to obtain near optimal performance for many choices of q and n. But you need a gap of about 5 bits between q and R.
- Not quite as fast as Longa and Naehrig for specific case of $q = 12289$. They exploit form of $q = 1 \bmod 2n$.
- Approximately 36K Intel i5 clock cycles for NTT or inverse NTT, 24-bit q, $n = 1024$
- Possible to implement NewHope-Simple in constant time with no **if** statements!
- Any Questions?