Implementation and Optimization of the Number Theoretic Transform

Bruno Dutertre
SRI International

Formal Methods Reading Group
February 23, 2017

Joint Work with Ian Mason and Tancrède Lepoint

Lattice-Based Cryptography

Lattices

- Linear combinations of vectors with integer coefficients:

$$L = \mathcal{L}(b_1, \ldots, b_t) = \left\{ \sum_{i=1}^{t} x_i b_i \mid x_i \in \mathbb{Z} \right\}$$

- Some problems are believed to be computationally hard:
  - SVP: find the shortest element in L
  - CVP: find a point of L closest to a given $x \in \mathbb{Z}^m$

Lattice-Based Crypto

- Cryptographic algorithms (public key encryption, digital signatures, homomorphic encryption, etc.), whose security relies on the hardness of lattice problems.
- As far as we know, they're secure against quantum computers
- They are becoming increasingly practical

Basic Setting

General q-ary Lattices

- q is a small prime number (e.g., $q = 12289$)
- Algorithms are based on linear algebra computations using matrices with coefficients in $\mathbb{Z}_q$
- Secret and public keys are matrices in $\mathbb{Z}_q^{n \times m}$

Practical Instantiations

- Use matrices with a special structure (e.g., anticirculant matrices)

$$A = \begin{bmatrix} a_0 & a_1 & \dots & a_n \\ -a_n & a_0 & \dots & a_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ -a_1 & -a_2 & \dots & a_o \end{bmatrix}$$

- This reduces key sizes and algorithms can be implemented using polynomial operations.

Example: Abstract BLISS

Algorithm 1: Signature Verification

Input : Message μ
Public key $A \in \mathbb{Z}_{2q}^{n \times m}$
Signature (z, c) where $z \in \mathbb{Z}^m$ and $c \in \mathbb{Z}^n$

Output: Accept or Reject

if $||z|| > B_2$ **then** Reject
if $||z||_\infty \geqslant q/4$ **then** Reject
if $c = H(Az + qc \bmod 2q, \mu)$ **then** Accept **else** Reject

Concrete BLISS

Algorithm 2: Signature Verification

Input : Message μ
Public key a_1 a polynomial of degree $\leqslant n-1$ in $\mathbb{Z}_q[X]/(X^n+1)$
Signature (z_1, z_2, c) triple of polynomials

Output: Accept or Reject

if $||(z_1|2^d z_2)|| > B_2$ **then** Reject
if $||(z_1|2^d z_2)||_\infty > B_\infty$ **then** Reject
if $c = H(\lfloor (2\zeta a_1.z_1 + \zeta q c) \bmod 2q \rceil_d + z_2 \bmod p, \mu)$ **then**
  Accept
else
  Reject
end

Anticirculant Matrices and Polynomials

Correspondence

- Given an anticirculant matrix
$$A = \begin{bmatrix} a_0 & a_1 & \ldots & a_n \\ -a_n & a_0 & \ldots & a_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ -a_1 & -a_2 & \ldots & a_o \end{bmatrix}$$
we can construct the polynomial
$$p(A) = a_0 + a_1 X + \ldots + a_{n-1} X^{n-1}$$
(so $p(A)$ is a polynomial a degree at most $n-1$, with coefficients in $\mathbb{Z}_q$).
- Conversely, we can map any polynomial p of this form to an anticirculant matrix $M(p)$.

Anticirculant Matrices and Polynomials

Nice Property

- The product of two anticirculant matrices is the same thing as the product of their polynomials in $\mathbb{Z}_q[X]/(X^n+1)$:

$$p(A \times B) = p(A).p(B)$$

$$M(p_1.p_2) = M(p_1) \times M(p_2)$$

Fast computation of products in $\mathbb{Z}_q[X]/(X^n+1)$ is then important in lattice-based crypto.

Products in $\mathbb{Z}_q[X]/(X^n + 1)$

Definition

- Given two polynomials f and g (of degree at most $n-1$):

$$\begin{aligned} f &= a_0 + a_1 X + \ldots + a_{n-1} X^{n-1} \\ g &= b_0 + b_1 X + \ldots + b_{n-1} X^{n-1} \end{aligned}$$

  then their product is $h = c_0 + c_1 X + \ldots + c_{n-1} X^{n-1}$, where

$$c_i = \Big(\sum_{j+k=i} a_j b_k - \sum_{j+k=n+i} a_j b_k\Big) \bmod q$$

Example in $\mathbb{Z}_{17}[X]/(X^4 + 1)$

$$\begin{aligned} (3 + X^3)(6X + X^2) &= 18X + 3X^2 + 6X^4 + X^5 \ = \ 18X + 3X^2 - 6 - X \\ &= -6 + 17X + 3X^2 \ = \ -6 + 3X^2. \end{aligned}$$

Primitive Roots of Unity in $\mathbb{Z}_q$

$\omega \in \mathbb{Z}_q$ is a primitive n-th root of unity if

- $\omega^n = 1$
- $\omega^i \neq 1$ for any i such that $0 < i < n$.

Example: 722 is a primitive 16-th root of unity in $\mathbb{Z}_q$ for $q = 12289$

Properties

- Primitive n-th roots of unity exist iff n divides $q-1$ (for q prime)
- If n is a power of two then ω is a primitive n-th root of unity iff $\omega^{n/2} = -1$.
- If ω is a primitive n-th root of unity and $s \in \mathbb{Z}$ then we have

$$\sum_{i=0}^{n-1} \omega^{si} = \begin{cases} n \text{ if } n \text{ divides } s \\ 0 \text{ otherwise} \end{cases}$$

The Number-Theoretic Transform

Definition

- We assume a fixed ω that's an n-th root of unity in $\mathbb{Z}_q$.
- Given $(a_0, \ldots, a_{n-1})$ in $\mathbb{Z}_q^n$ then $\mathrm{NTT}(a_0, \ldots, a_{n-1})$ is the tuple $(\tilde{a}_0, \ldots, \tilde{a}_{n-1})$ defined by

$$\tilde{a}_i = \sum_{j=0}^{n-1} a_j \omega^{ij}.$$

- If we define $f(X) = a_0 + a_1 X + \ldots + a_{n-1} X^{n-1}$ then $\tilde{a}_i$ is just $f(\omega^i)$.

Note this is bijection:

- Given any $(\tilde{a}_0, \ldots, \tilde{a}_{n-1}) \in \mathbb{Z}_q^n$, there's a unique tuple $(a_0, \ldots, a_{n-1}) \in \mathbb{Z}_q^n$ such that $\mathrm{NTT}(a_0, \ldots, a_{n-1}) = (\tilde{a}_0, \ldots, \tilde{a}_{n-1})$.

Inverse Transform

Inverse

- Since ω is an n-th root of unity, ω^{-1} is one too.
- Given any $(\tilde{a}_0, \ldots, \tilde{a}_{n-1})$ in $\mathbb{Z}_q^n$, we define $\mathrm{INTT}(\tilde{a}_0, \ldots, \tilde{a}_{n-1}) = (a'_0, \ldots, a'_{n-1})$ where

$$a'_i \;=\; n^{-1} \sum_{j=0}^{n-1} \tilde{a}_j \omega^{-ij} \;=\; n^{-1} f(\omega^{-i}).$$

- Then we have

$$\begin{aligned}
\mathrm{INTT}(\mathrm{NTT}(a_0, \ldots, a_{n-1})) &= (a_0, \ldots, a_{n-1}) \\
\mathrm{NTT}(\mathrm{INTT}(a_0, \ldots, a_{n-1})) &= (a_0, \ldots, a_{n-1}).
\end{aligned}$$

  NTT and INTT are inverses of each other.

Note: NTT is almost its own inverse:

$$\mathrm{NTT}(\mathrm{NTT}(a_0, \ldots, a_{n-1})) \;=\; n \times (a_0, a_{n-1}, \ldots a_1).$$

NTT and Products of Polynomials

Products in $\mathbb{Z}_q[X]$

- If f and g have degree less than $m = n/2$:

$$\begin{aligned} f &= a_0 + a_1 X + \ldots + a_{m-1} X^{m-1} \\ g &= b_0 + b_1 X + \ldots + b_{m-1} X^{m-1} \end{aligned}$$

their product $h = fg$ has degree less than n:

$$h = c_0 + c_1 X + \ldots + c_{n-1} X^{n-1}$$

NTT-Based Computation

- Compute

$$\begin{aligned} (\tilde{a}_0, \ldots, \tilde{a}_{n-1}) &= \mathrm{NTT}(a_0, \ldots, a_{m-1}, 0, \ldots, 0) \\ (\tilde{b}_0, \ldots, \tilde{b}_{n-1}) &= \mathrm{NTT}(b_0, \ldots, b_{m-1}, 0, \ldots, 0) \end{aligned}$$

- Compute the element-wise products: $\tilde{c}_i = \tilde{a}_i \tilde{b}_i$
- Then we have $(c_0, \ldots, c_{n-1}) = \mathrm{INTT}(\tilde{c}_0, \ldots, \tilde{c}_{n-1})$

Products in $\mathbb{Z}_q[X]/(X^n + 1)$

Issue

- The previous approach does not work in $\mathbb{Z}_q[X]/(X^n + 1)$ in general.
- For example, we may have $(f.g)(\omega^i) \neq f(\omega^i)g(\omega^i)$

Solution

- Introduce a new constant ψ such that $\psi^2 = \omega$.
- Then ψ is a primitive $2n$-th root of unity, so $2n$ must divide $q - 1$.
- Then one can compute $f.g$ by multiplying by powers of ψ and ψ^{-1}

Products in $\mathbb{Z}_q[X]/(X^n+1)$

Preprocess

- Multiply all coefficients of f and g by powers of ψ:

$$a'_i \;=\; a_i\psi^i \quad \text{and} \quad b'_i = b_i\psi^i$$

NTT Steps

$$\begin{aligned}
(\tilde{a}'_0,\ldots,\tilde{a}'_{n-1}) &= \mathrm{NTT}(a'_0,\ldots,a'_{n-1}) \\
(\tilde{b}'_0,\ldots,\tilde{b}'_{n-1}) &= \mathrm{NTT}(b_0,\ldots,b'_{n-1}) \\
\tilde{c}_i &= \tilde{a}_i\tilde{b}_i \qquad \text{for } i=0,\ldots,n-1 \\
(c'_0,\ldots,c'_{n-1}) &= \mathrm{INTT}(\tilde{c}_0,\ldots,\tilde{c}_{n-1})
\end{aligned}$$

Post processing

- Multiply by powers of ψ^{-1}:

$$(c_0,c_1,\ldots,c_{n-1}) \;=\; (c'_0,c'_1\psi^{-1},\ldots,c'_{n-1}\psi^{-(n-1)})$$

Fast NTT Computation

Input

- $f = a_0 + a_1 X + \ldots + a_{n-1} X^{n-1}$ polynomial
- ω: primitive n-th root of unity
- q is prime and n is a power of two

Output

$$\begin{aligned} \tilde{a}_0 &= f(1) \\ \tilde{a}_1 &= f(\omega) \\ &\vdots \\ \tilde{a}_{n-1} &= f(\omega^{n-1}) \end{aligned}$$

There are algorithms for computing this with $O(n \log n)$ multiplications in $\mathbb{Z}_q$

The Cooley-Tukey Method

Idea

- Write $f(X)$ as $g(X^2) + Xh(X^2)$ by splitting even and odd powers of X:

$$
\begin{aligned}
g(X) &= a_0 + a_2 X + \ldots + a_{n-2} X^{n/2-1} \\
h(X) &= a_1 + a_3 X + \ldots + a_{n-1} X^{n/2-1}
\end{aligned}
$$

- Recursively compute the NTT of g and h, using ω^2 as $n/2$-th root of unity.

- From the results of these two subproblems, compute the coefficients $\tilde{a}_i = f(\omega^i)$.

Cooley-Tukey Details

Main Step

- The recursive computations give $g(1), g(\omega^2), \ldots, g(\omega^n)$ and $h(1), h(\omega^2), \ldots, h(\omega^n)$
- To compute $\tilde{a}_i$ from them:
  - For $0 \leqslant i < n/2$, we have

$$\tilde{a}_i \;=\; f(\omega^i) = g(\omega^{2i}) + \omega^i h(\omega^{2i})$$

  - For $n/2 \leqslant i < n$, we set $j = i - n/2$, and we get

$$\begin{aligned} \tilde{a}_i &= g(\omega^{2i}) + \omega^i h(\omega^{2i}) \\ &= g(\omega^{n+2j}) + \omega^{j+n/2} h(\omega^{n+2j}) \\ &= g(\omega^{2j}) - \omega^j h(\omega^{2j}) \end{aligned}$$

    because $\omega^n = 1$ and $\omega^{n/2} = -1$.

The Gentleman-Sande Method

Idea

- By definition of NTT, we want to compute n sums of n terms:

$$\tilde{a}_i \;=\; \sum_{j=0}^{n-1} a_j \omega^{ij}.$$

- Split the problem into two subproblems of size $n/2$

$$\tilde{a}_{2k} \;=\; \sum_{j=0}^{n/2-1} b_j \omega^{2kj} \quad \text{and} \quad \tilde{a}_{2k+1} \;=\; \sum_{j=0}^{n/2-1} c_j \omega^{2kj}$$

- Recursively solve the subproblems to get $\tilde{b}_0, \ldots, \tilde{b}_{n/2-1}$ and $\tilde{c}_0, \ldots, \tilde{c}_{n/2-1}$ and we're done:

$$\begin{aligned} \tilde{a}_{2k} &= \tilde{b}_k \;\text{ for } k = 0, \ldots, n/2-1 \\ \tilde{a}_{2k+1} &= \tilde{c}_k \;\text{ for } k = 0, \ldots, n/2-1 \end{aligned}$$

Gentleman-Sande Details

Deomposition Into Two Subproblems

- For each coefficient $\tilde{a}_i$, we split the sum into two halves:

$$
\begin{aligned}
\tilde{a}_i &= \sum_{j=0}^{n/2-1} a_j \omega^{ij} + \sum_{j=n/2}^{n-1} a_j \omega^{ij} \\
&= \sum_{j=0}^{n/2-1} a_j \omega^{ij} + a_{n/2+j} \omega^{i(n/2+j)} \\
&= \sum_{j=0}^{n/2-1} (a_j + a_{n/2+j} \omega^{in/2}) \omega^{ij}
\end{aligned}
$$

Gentleman-Sande Details (continued)

Decomposition Into Two Subproblems

- For $i = 2k$, we have $\omega^{in/2} = \omega^{nk} = 1$, so

$$\tilde{a}_{2k} = \sum_{j=0}^{n/2-1} (a_j + a_{n/2+j})\omega^{2kj}$$

- For $i = 2k+1$, we have $\omega^{in/2} = \omega^{nk+n/2} = -1$, so

$$\begin{aligned}\tilde{a}_{2k+1} &= \sum_{j=0}^{n/2-1} (a_j - a_{n/2+j})\omega^{(2k+1)j} \\ &= \sum_{j=0}^{n/2-1} ((a_j - a_{n/2+j})\omega^{j})\omega^{2kj}\end{aligned}$$

- Two subproblems are given by $(b_0, \ldots, b_{n/2-1})$ and $(c_0, \ldots, c_{n/2-1})$ where

$$\begin{aligned}b_j &= a_j + a_{n/2+j} \\ c_j &= (a_j - a_{n/2+j})\omega^j\end{aligned}$$

Real Implementation

Efficient, Non-recursive Algorithms

- All computations are done in place and update an array of n integers
- All the coefficients ω^i are precomputed and stored in a constant table
- Multiplication of the input by powers of ψ can be integrated into the Cooley-Tukey algorithm
- Multiplication of the output by powers of ψ^{-1} can be integrated into the Gentleman-Sande algorithm

Bitreversed Permutations

Side effect:

- If the input is in standard order, the in-place updates cause the output to be produced in *bitreversed* order:
  - Input: coefficient a_i stored at index i
  - Output: coefficient $\tilde{a}_i$ stored at index $\mathrm{bitrev}_k(i)$
- Alternatively, we can have input in bitreverse order and output in standard order
  - Input: coefficient a_i stored at index $\mathrm{bitrev}_k(i)$
  - Output: coefficient $\tilde{a}_i$ stored at index i

  where $\mathrm{bitrev}_k(i)$ takes the k-bit representation of j and reverses all the bits.

  For example $\mathrm{bitrev}_4(5) = \mathrm{bitrev}_4(0\mathrm{b}0101) = 0\mathrm{b}1010 = 10$.

Example C Code

```
#define Q 12289

void mulntt_ct_rev2std(int32_t *a, uint32_t n, const uint16_t *p) {
  uint32_t j, s, t;
  int32_t x, w;

  for (t=1; t<n; t <<= 1) {
    for (j=0; j<t; j++) {
      w = p[t + j]; // w = psi_t * omega_t^j
      for (s=j; s<n; s += t + t) {
        x = a[s + t] * w;
        a[s + t] = (a[s] - x) % Q;
        a[s] = (a[s] + x) % Q;
      }
    }
  }
}
```

Accelerating Reductions Modulo q

Issue

- General reduction modulo q requires integer division, which is very slow.
  On Intel Haswell, a 32bit `DIV` has a latency of 22-29 clock ticks, that's about 10 times slower than a 32bit `MUL`

Possible Optimizations

- Reduce the number of mod q operations (e.g., do them lazily)
- Replace mod q operations by faster code since q is a fixed constant (e.g., Harvey, 2013)
- Montgomery's encoding (Montgomery, 1983)

Example Low-level Optimizations

```
// (x - y) % Q
static inline int32_t sub_mod(int32_t x, int32_t y) {
  x -= y;
  return x + ((x >> 14) & Q);
}

// (x + y) % Q
static inline int32_t add_mod(int32_t x, int32_t y) {
  x += y - Q;
  return x + ((x >> 14) & Q);
}

// integer division assuming 0 <= x < (Q-1)^2
static inline uint32_t divq(int32_t x) {
  return (((uint64_t) x) * 178942409) >> 41;
}

// mod
static inline int32_t modq(int32_t x) {
  return x - divq(x) * Q;
}
```

Improved C Code

```
void mulntt_ct_rev2std(int32_t *a, uint32_t n, const uint16_t *p) {
  uint32_t j, s, t;
  int32_t x, w;

  for (t=1; t<n; t <<= 1) {
    for (j=0; j<t; j++) {
      w = p[t + j]; // w = psi_t * omega_t^j
      for (s=j; s<n; s += t + t) {
        x = modq(a[s + t] * w);  // (a[s+t] * w) mod Q
        a[s + t] = sub_mod((a[s], x);
        a[s] = add_mod(a[s], x);
      }
    }
  }
}
```

Longa and Naehrig's Reduction, 2016

Properties of the Modulus q

- We know that $2n$ divides $q-1$ and $n = 2^t$ is a power of two, so we can write q as

$$q \;=\; k.2^m + 1$$

  where m is at least $t+1$ and k is an odd number.
- For well-chosen q, the constant k is small. For example $12289 = 3.2^{12} + 1$.

Reduction

- For an integer $x \in \mathbb{Z}$

$$\mathrm{red}(x) \;=\; k.(x \bmod 2^m) - (x/2^m)$$

- That's cheap to compute:

```
static inline int32_t red(int32_t x) {
  return 3 * (x & 4095) - (x >> 12);
}
```

Properties of the Reduction

Main Property: $\mathrm{red}(x) = kx \bmod q$

- We can replace $x \bmod q$ by $\mathrm{red}(x)$ in the algorithms
- We're off by a factor of k but in most cases we can precompute a correction
- Example in Cooley Tukey
  - We evaluate $\omega^i a_{s+t} \bmod q$ where ω^i is precomputed and stored in a table
  - We can precompute $c = k^{-1}\omega^i \bmod q$ instead and then

$$\mathrm{red}(ca_{s+t}) \;=\; kca_{s+t} \bmod q \;=\; kk^{-1}\omega^i a_{s+t} \bmod q \;=\; \omega^i a_{s+t} \bmod q$$

Second Property: $|\mathrm{red}(cx)|$ grows slowly

$$0 \leqslant c \leqslant q-1 \;\Rightarrow\; |\mathrm{red}(cx)| \leqslant k|x| + (q-k)$$

Only Issue: Prevent Numerical Overflows

Risk of Overflows

- Typical parameters: q is 12289 ($k = 3$ and $m = 12$) and $n = 512$ or $n = 1024$
- The NTT computations require then 9 or 10 iterations
- The input can be assumed to be small $|a_i| \leqslant q - 1$
- Even though the elements $a[i]$ grow slowly, there maybe a risk of overflow after 8 iterations for CT or 6 iterations for GS

To Prevent This

- Apply one round of reduction to all elements of array a after 7 or 5 iterations

Example C Code

```c
void mulntt_red_ct_rev2std(int32_t *a, uint32_t n, const int16_t *p) {
  uint32_t j, s, t;
  int32_t x, w;

  for (t=1; t<n; t <<= 1) {
    if (t == 128) {
      // prevent overflow
      for (j=0; j<n; j++) {
        a[i] = red(a[i]);
      }
    }
    // normal algorithm
    for (j=0; j<t; j++) {
      w = p[t + j]; // w = psi_t * w_t^j * inverse(3)
      for (s=j; s<n; s += t + t) {
        x = mul_red(a[s + t], w);
        a[s + t] = a[s] - x;
        a[s] = a[s] + x;
      }
    }
  }
}
```

Improvements to Longa & Naehrig's Implementation

More Precise Analysis

- The bounds used by Longa & Naehrig to estimate $|a[i]|$ can be made more precise
- The $|a[i]|$s grow more slowly than what Longa & Naehrig computed

Use Negative Coefficients

- The algorithms used tables of constants in $[0, q-1]$
- It's better to shift these constants to $[-\frac{q}{2}, \frac{q}{2} - 1]$
- If we do that, there's no risk of overflow for $n \leqslant 1024$.

Current Status

GitHub Repository

- All our code is at `https://github.com/SRI-CSL/Bliss`
- This includes an implementation of the BLISS-B signature scheme and several NTT computations written in C

On-going Work

- NTT implementations in assembler to use vector instructions of Intel x86-64 (AVX2)

Future Work

Abstract Interpretation

- Computing bounds on $|\mathrm{red}(a_i c)|$ by hand is tedious and error-prone
- The hand-computed bounds are always somewhat imprecise (pessimistic)
- We can get exact bounds by using abstract interpretation and interval analysis
- This should lead to more efficient and higher-assurance implementations

Synthesis/Compilation

- It should be possible to take automatically rewrite a procedure that uses mod q operations into an equivalent procedure that uses the reduction approach, with appropriate corrections, and reductions to prevent numerical overflows

Acknowledgments

Thanks to Ulf & Pat for IRAD funding for this work

Thanks to Ian and Tancrède