Verification of an Optimized NTT Algorithm

Jorge A. Navas, Bruno Dutertre, Ian A. Mason
SRI International

VSTTE
July 21, 2020

Lattice-Based Cryptography

Current Public-Key Cryptography

- Based on hardness of factorization or computing discrete logarithms
- These can be broken by quantum computers

Lattice-Based Cryptography

- Based on hardness of problems related to finding small vectors in lattices
- Quantum Resistant: No known quantum algorithm solves these problems
- Supports new constructs such as fully homomorphic encryption
- Increasingly practical

A Common Procedure in Lattice-Based Crypto

Polynomial Ring

- We work in $\mathbb{Z}_q[X]/(X^n+1)$ where q is a prime number
- Polynomials are of the form $a_0 + a_1 X + \ldots + a_{n-1}X^{n-1}$ (degree at most $n-1$) where the $a_i \in \mathbb{Z}_q$ are integers modulo q.

Product of Polynomials

- Given

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

their product is

$$f.g \;=\; c_0 + c_1 X + \ldots + c_{n-1}X^{n-1}$$

where

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

The Number-Theoretic Transform

Primitive Roots of Unity

- $\omega \in \mathbb{Z}_q$ is a primitive n-th root of unity if $\omega^n = 1$ and $\omega^i \neq 1$ for any i such that $0 < i < n$.
- Primitive n-th roots of unity exist iff n divides $q - 1$.

Number-Theoretic Transform

- Assume a fixed ω that's a primitive root of unity
- Given a tuple $a = (a_0, \ldots, a_{n-1})$ in $\mathbb{Z}_q^n$ and $f = a_0 + a_1 X + \ldots a_{n-1} X^{n-1}$
- Forward transform: $\mathrm{NTT}(a)$ is the tuple $\tilde{a} = (\tilde{a}_0, \ldots, \tilde{a}_{n-1})$ defined by

$$\tilde{a}_i \;=\; \sum_{j=0}^{n-1} a_j \omega^{ij} \bmod q \;=\; f(\omega^i)$$

Inverse Transform

Inverse

- Since ω is an n-th root of unity, ω^{-1} is one too.
- Inverse transform: $\mathrm{INTT}(a_0, \ldots, a_{n-1}) = (a'_0, \ldots, a'_{n-1})$ where

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

- Then NTT and INTT are inverses of each other:

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

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

Input: $f = a_0 + a_1 X + \ldots + a_{n-1} X^{n-1}$
$g = b_0 + b_1 X + \ldots + b_{n-1} X^{n-1}$

Output: $h = c_0 + c_1 X + \ldots + c_{n-1} X^{n-1}$ such that $h = f.g$.

Procedure:

$$\begin{aligned}
\hat{a} &:= (a_0, a_1\psi, \ldots, a_{n-1}\psi^{n-1}) \\
\hat{b} &:= (b_0, b_1\psi, \ldots, b_{n-1}\psi^{n-1}) \\
\tilde{a} &:= \mathrm{NTT}(\hat{a}) \\
\tilde{b} &:= \mathrm{NTT}(\hat{b}) \\
\tilde{c} &:= (\tilde{a}_0\tilde{b}_0, \ldots, \tilde{a}_{n-1}\tilde{b}_{n-1}) \\
\hat{c} &:= \mathrm{INTT}(\tilde{c}) \\
c &:= (\hat{c}_0, \hat{c}_1\psi^{-1}, \ldots, \hat{c}_{n-1}\psi^{-(n-1)})
\end{aligned}$$

where ψ is a new parameter such that $\psi^2 = \omega$.

Fast NTT Computation

```
// Cooley-Tukey variant
void ntt_ct_std2rev(int32_t *a, uint32_t n,
     const uint16_t *p) {
  uint32_t j, s, t, u, d;
  int32_t x, w;

  d = n;
  for (t = 1; t < n; t <<= 1) {
    d >>= 1;
    u = 0;
    for (j = 0; j < t; j++) {
      w = p[t + j]; // w_t^bitrev(j)
      u += 2 * d;
      for (s = u; s < u + d; s++) {
        x = a[s + d] * w;
        a[s + d] = (a[s] - x) % Q;
        a[s] = (a[s] + x) % Q;
      }
    }
  }
}
```

```
// Gentleman-Sande Variant
void ntt_gs_std2rev(int32_t *a, uint32_t n,
     const uint16_t *p) {
  uint32_t j, s, t;
  int32_t w, x;

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

- Specialized for the case where n is a power of two
- Need $O(n \log n)$ multiplications in $\mathbb{Z}_q$
- Powers of ω are precomputed and stored in array p

Accelerating Reductions Modulo q

Issue

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

Possible Optimizations

- Reduce the number of mod q operations (Harvey, 2013)
- Replace mod q operations by faster code since q is a known constant (Warren, 2013)
- Montgomery's reduction, 1983
- Longa and Naehrig's reduction, 2016

Longa and Naehrig's Reduction

Properties of the Modulus q

- We know that $2n$ divides $q-1$ and $n = 2^t$ is a power of two
- We can write $q = k.2^m$ where $m \geqslant t+1$ and k is an odd number.
- For well-chosen q, the constant k is small. A common choice is $q = 12289 = 3.2^{12} + 1$.

Reduction

- For an integer $x \in \mathbb{Z}$: $\mathsf{red}(x) \;=\; k \times (x \bmod 2^m) - \lfloor x/2^m \rfloor$
- Properties
  - Main property: $\mathsf{red}(x) = kx \pmod{q}$
  - Slow growth: $0 \leqslant c \leqslant q-1 \;\Rightarrow\; |\mathsf{red}(cx)| \leqslant k|x| + (q-k)$
  - Cheap to compute

Example Fast NTT with Longa/Naehrig

```c
// red(x * y) when Q=12289
static int32_t mul_red(int32_t x, int32_t y) {
  int64_t z;
  z = (int64_t) x * y; x = z & 4095; y = z >> 12;
  return 3 * x - y;
}

// Cooley-Tukey variant
void ntt_red_ct_std2rev(int32_t *a, uint32_t n, const int16_t *p) {
  uint32_t j, s, t, u, d;
  int32_t x, y, w;
  d = n;
  for (t = 1; t < n; t <<= 1) {
    d >>= 1;
    u = 0;
    for (j = 0; j < t; j++) {
      w = p[t + j]; // k^(-1) w_t^bitrev(j)
      u += 2 * d;
      for (s = u; s < u + d; s++) {
        y = a[s];
        x = mul_red(a[s + d], w);
        a[s] = y + x;
        a[s + d] = y - x;
      }
    }
  }
}
```

Main issue: Show that there are no integer overflows

Verification

Goals

- Check absence of overflows for $n = 1024$ (also $n = 512$ and $n = 2048$)
- Assumptions on input: $0 \leqslant \texttt{a[}i\texttt{]} \leqslant 12288$ for $i = 0, \ldots, n-1$.
- All constants in $\texttt{p}$ satisfy $-6144 \leqslant \texttt{p[}i\texttt{]} \leqslant 6144$.

Static Analysis Tools we Tried

- Bounded model checking: CBMC does not scale
- Symbolic execution: SAW/Crucible does not scale
- Software model checkers: CPACheckers and SeaHorn/PDR timeout
- Abstract interpretation: SeaHorn/CRAB failed to prove property

Main Issues

- The red function involves bit-shift and masks
- Complex loops and array indexing

Specialized Approach

Abstract Domain: Intervals

- Extended with a transfer function for the Longa-Naehrig reduction:

$$\mathsf{red}_{\mathcal{I}}([a,b]) \;=\; [\mathsf{red}(\max(b \,\&\, {\sim}4095, a)), \mathsf{red}(\min(a \mid 4095, b))]$$

Array and Indices

- Array domains we tried are too imprecise
- Solution: for a fixed n, all the loops in the NTT procedures are bounded. We just unroll these loops.

SeaHorn Implementation

- Extended CRAB with the new transfer function
- Modified SeaHorn to handle red as an LLVM intrisic
- Use LLVM opt for loop unrolling

SeaHorn Results

		Time (sec)	
Program	Description	Basic	LLVM opt
intt_red1024	inv CT/std2rev, $\psi = 1014$	900	9
intt_red1024b	inv CT/rev2std, $\psi = 1014$	972	9
ntt_red1024	CT/std2rev, $\psi = 1014$	923	9
ntt_red1024b	CT/rev2std, $\psi = 1014$	836	9
ntt_red1024c	CT/std2rev	1151	9
ntt_red1024d	CT/rev2std	1258	11
ntt_red1024e	GS/std2rev, $\psi = 1014$	8265	10
ntt_red1024f	GS/rev2std, $\psi = 1014$	8115	10

Two versions

- Basic: extended interval domain + loop unrolling
- LLVM Optimized: aggressive preprocessing with LLVM: inlining and scalar replacement of aggregates (SROA)

Alternative Implementation: Source-Code Rewriting

```
// red_scale(w, a) returns an interval [l, h]
// such that l <= red(w * x) <= h for any x in a
extern interval_t *red_scale(int64_t w, const interval_t *a);

// ntt_red_ct_std2rev modified to operate in the abstract domain (i.e., on intervals)
void abstract_ntt_red_ct_std2rev(interval_t **a, uint32_t n, const int16_t *p) {
  uint32_t j, s, t, u, d;
  interval_t *x, *y, *z;
  int64_t w;
  d = n;
  for (t = 1; t < n; t <<= 1) {
    show_intervals("ct_std2rev", t, a, n); // print intervals and check for overflow
    d >>= 1;
    u = 0;
    for (j = 0; j < t; j++) {
      w = p[t + j]; // w_t^bitrev(j) extended to 64 bits
      u += 2 * d;
      for (s = u; s < u + d; s++) {
        x = a[s + d];
        y = a[s];
        z = red_scale(w, x);
        a[s + d] = sub(y, z);
        a[s] = add(y, z);
      }
    }
  }
  show_intervals("ct_std2rev", t, a, n);
}
```

Code-Rewriting Results

Program	Description	Time (sec)
intt_red1024	inv CT/std2rev, $\psi = 1014$	0.02
intt_red1024b	inv CT/rev2std, $\psi = 1014$	0.02
ntt_red1024	CT/std2rev, $\psi = 1014$	0.02
ntt_red1024b	CT/rev2std, $\psi = 1014$	0.02
ntt_red1024c	CT/std2rev	0.56
ntt_red1024d	CT/rev2std	0.56
ntt_red1024e	GS/std2rev, $\psi = 1014$	0.21
ntt_red1024f	GS/rev2std, $\psi = 1014$	0.19

More Results and Notes

Generalizations

- For $n = 1024$, the NTT procedures we verified are safe under weaker assumptions (we can use larger bounds on the input).
- For $n = 1024$ we have a generic proof that just assume bounds on $\texttt{p}[i]$ (ntt_red1024c and ntt_red1024d in the tables)
- For $n = 2048$, the generic proof fails
- But the procedures are safe for $n = 2048$ (we prove this by exhaustively checking every possible value of ψ)

Beyond Static Analysis

- Very hard to get precise enough bounds on $\texttt{a}[i]$ by hand
- This limits applicability of deductive methods

Conclusion

Number-Theoretic Transforms

- Used in practical quantum-resilient cryptographic schemes
- Many optimization proposed to make it faster
- About 10-20 lines of code but challenging for static analysis tools
- Our solution: loop unrolling, interval analysis, specialized transfer function

Future Work

- Full correctness of NTT procedures and polynomial products

Links

- Examples and results: `https://github.com/SRI-CSL/NTT`
- SeaHorn: `https://github.com/seahorn/seahorn`
- CRAB/Clam: `https://github.com/seahorn/crab`, `https://github.com/seahorn/crab-llvm`