Verification of an Optimized NTT Algorithm

Jorge A. Navas, Bruno Dutertre, and Ian A. Mason

Computer Science Laboratory, SRI International, Menlo Park CA 94025, USA
firstname.lastname@sri.com

Abstract. The Number Theoretic Transform (NTT) is an efficient algorithm for computing products of polynomials with coefficient in finite fields. It is a common procedure in lattice-based key-exchange and signature schemes. These new cryptographic algorithms are becoming increasingly important because they are *quantum resistant*. No quantum algorithm is known to break these lattice-based algorithms, unlike older schemes such as RSA or elliptic curve cryptosystems.

Many implementations and optimizations of the NTT have been proposed in the literature. A particular efficient variant is due to Longa and Naehrig. We have implemented several of these variants, including an improved version of the Longa and Naehrig algorithm. An important concern is to show that numerical overflows do not happen in such algorithms. We report on several attempts at automatically verifying the absence of overflows using static analysis tools. Off-the-shelf tools do not work on the NTT code. We present a specialized abstract-interpretation method to solve the problem.

1 Introduction

We present an experiment in verification of an optimized implementation of the Number Theoretic Transform (NTT). This transform is a key procedure in lattice-based cryptography, one of the most promising approach to developing quantum-resistant replacement for today's public-key cryptography. Current schemes are based on the hardness of factoring or discrete logarithms and will be broken if or when quantum computers become practical.

We first give an overview of the NTT and its application to computing products of polynomials. We then discuss several optimizations that attempt to reduce the cost of modular operations. A particular efficient method is due to Longa and Naehrig [29]. We propose to further improve their algorithm, but correctness of this improvement requires showing that no integer overflow is possible (when implemented using 32bit integers).

We discuss our attempts at proving that no such overflow occurs by using different software verification tools and methods. Because the procedures use combinations of array manipulation, arithmetic, shift, and bit-masking, they are difficult to prove correct with off-the-shelf tools. We present a specialized abstract interpretation method that can solve this problem. The method is implemented in SeaHorn [24] and uses the Crab abstract interpretation library [12]. We also describe an alternative technique that relies on source-code transformation.

2 The Number Theoretic Transform

Lattice-based cryptography is based on the hardness of problems such as finding a short vector in an integer lattice. Commonly used lattices are defined by matrices $A \in \mathbb{Z}_q^{n \times m}$ where q is a prime number and $\mathbb{Z}_q$ denotes the ring of integers modulo q. Efficient implementations use lattices with a special structure that allows large random matrices to be replaced by random polynomials in $\mathbb{Z}_q[X]/(X^n+1)$. One of the most common operations is computing the product of two such polynomials: given $f = a_0 + \ldots + a_{n-1}X^{n-1}$ and $g = b_0 + \ldots + b_{n-1}X^{n-1}$, their product is the polynomial $h = c_0 + \ldots + c_{n-1}X^{n-1}$ defined by

$$c_i = \left(\sum_{j+k=i} a_j b_k - \sum_{j+k=n+i} a_j b_k \right) \bmod q. \tag{1}$$

The Number Theoretic Transform (NTT) is a specialization of the Fast Fourier Transform for computing such products.

The NTT starts with a number ω that is a primitive n-th root of unity in $\mathbb{Z}_q$. This means that $\omega^n = 1 \pmod q$ and that $\omega^m \neq 1 \pmod q$ when $0 < m < n$. Such an ω exists as long as n divides $q-1$ (for q prime). Let $a = (a_0, \ldots, a_{n-1})$ be a vector of n elements in $\mathbb{Z}_q$, and let $f = a_0 + \ldots + a_{n-1}X^{n-1}$, then the *forward transform* of a, denoted by $\mathrm{NTT}(a)$, is the vector $\tilde{a} = (\tilde{a}_0, \ldots, \tilde{a}_{n-1})$ such that

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

The NTT is a bijection from $\mathbb{Z}_q^n$ to $\mathbb{Z}_q^n$. Its inverse INTT is given by $\mathrm{INTT}(\tilde{a}) = (b_0, \ldots, b_{n-1})$ where

$$b_i = n^{-1} \sum_{j=0}^{n-1} \tilde{a}_j \omega^{-ji} \bmod q \; = \; n^{-1} f(\omega^{-i}),$$

and we have $\mathrm{INTT}(\mathrm{NTT}(a)) = a$.

The vector c defined by Equation 1 is the *negative wrapped convolution* of a and b. A standard method for computing c is shown Figure 1. It requires an additional parameter ψ such that $\psi^2 = \omega \pmod q$ (thus, ψ is a $2n$-th primitive root of unity). The procedure first multiplies a and b by powers of ψ to form vectors $\hat{a} = (a_0, a_1\psi, \ldots, a_{n-1}\psi^{n-1})$ and $\hat{b} = (b_0, b_1\psi, \ldots, b_{n-1}\psi^{n-1})$. It then computes $\mathrm{NTT}(\hat{a})$ and $\mathrm{NTT}(\hat{b})$, multiplies the results component-wise, applies the inverse transform, and multiplies the result by powers of ψ^{-1}. This method is presented by Winkler [39] and it is a basic procedure in many implementations of lattice-based cryptographic algorithms (e.g. [34,36,29]).

A direct computation of products using Equation 1 has cost $O(n^2)$. The main benefit of the NTT is to reduce this cost to $O(n \log n)$, since both NTT and INTT can be implemented in $O(n \log n)$. This reduction is significant in lattice-based

```
Input:   $f = a_0 + a_1X + \ldots + a_{n-1}X^{n-1}$
         $g = b_0 + b_1X + \ldots + b_{n-1}X^{n-1}$

Output: $h = c_0 + c_1X + \ldots + c_{n-1}X^{n-1}$ such that $h = f.g$.

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

Fig. 1. NTT-Based Product of Polynomials in $\mathbb{Z}_q[X]/(X^n + 1)$.

cryptography because the polynomials are dense and n is relatively large (e.g., $n = 1024$ or $n = 512$ are commonly used).

In the rest of this paper, we fix the parameters q to 12289, which is the prime used in existing schemes such as Bliss [19,18] and New Hope [1]. With this choice of q, NTT with n as large as 2048 are possible, but we are mostly interested in the case $n = 1024$, which is used by Bliss and New Hope.

2.1 Basic NTT Implementation

Two possible implementations of the NTT are shown in Figure 2. The two procedures operate on an array `a` of `n` elements, where `n` is a power of two. Each function uses an auxiliary array `p` of pre-computed constants. For both functions, elements of `p` are powers of ω modulo q (so they can be stored as 16bit integers). On entry to either function, array `a` contains a vector of `n` integers in the range $[0, q)$. On exit, the array `a` stores NTT(a) in *bit-reversed order*: the i-th coefficient of NTT(a) is stored in `a[`j`]` where j is obtained by writing i in binary and reversing the bits. For example, if $n = 2^6 = 64$ and $i = 5$ then $j = \mathsf{bitrev}(i) = \mathsf{bitrev}(0b000101) = 0b101000 = 40$. Implementations of the inverse NTT are very similar to these two procedures.

Variants of the procedures in Figure 2 take input in bit-reversed order and produce results in the standard order. Many optimizations of the basic algorithms are possible. A simple one is to omit multiplications by `w` in lines 15 and 32 when `j=0`. But the most expensive operation involved in NTT computations is the reduction modulo q that occurs in the inner loops of both procedures. On typical Intel/AMD processors, the integer division instructions with 32bit operands commonly have a latency about 10 times larger than an integer multiply on the same size [20]. Thus many optimizations focus on removing integer divisions or replacing them with more efficient arithmetic. Harvey [26] presents several such optimizations for the inner loops of NTT and INTT. Other optimizations avoid pre-multiplications by powers of ψ and post-multiplications by powers ψ^{-1} by adjusting the pre-computed constants in the array `p` [36,35].

```c
#define Q 12289

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;
      }
    }
  }
}

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;
      }
    }
  }
}
```

Fig. 2. Two Variant NTT Implementations. The two procedures compute NTT(a) in place. Parameter `n` is the size of array `a`; it must be a power of 2. Both procedures return the result in array `a` in "bit-reversed" order. The computations use arrays `p` of pre-computed constants, which are all powers of ω. The top procedure follows Cooley-Tukey [7] and the bottom procedure uses the Gentleman-Sande variant [21].

One should note that compiler optimizations replace integer divisions by more efficient instruction sequences when the divisor is a known constant. In our case, $q = 12289$, and the unsigned 32bit division by q can be removed by using the equality $x \bmod q \; = \; x - \lfloor x/q \rfloor \; = \; x - \lfloor 2863078533x/2^{45} \rfloor$, which holds when $0 \leq x < 2^{32}$. In this equation, 2863078533 is $2^{45}/q$ suitably rounded. This optimization is applied by both GCC and Clang when compiling for x86-64. The resulting machine code uses three instructions but it is much more efficient than a single DIV instruction. These division tricks are explained in Chapter 10 of Warren's Hacker's Delight [38].

2.2 Longa and Naehrig's Reduction

Another type of optimization replaces reduction modulo q by a related operation that is cheaper to compute, such as Montgomery's reduction [32]. Instead of computing $x \bmod q$, the Montgomery reduction returns $y \equiv \alpha x \pmod{q}$ for some fixed constant α (i.e., α is the inverse of 2^{32} modulo q). One can easily correct for the extra factor α by adjusting the constants in array `p`. Although this reduction removes division by q, it is more expensive than the compiler optimization trick presented previously (Montgomery's reduction uses two multiplications).

```c
// single reduction
static int32_t red(int32_t x) {
  return 3 * (x & 4095) - (x >> 12);
}

// reduction of x * y using 64bit arithmetic
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;
}
```

Fig. 3. Longa-Naehrig reduction for $q = 12289 = 3.2^{12} + 1$

Longa and Naehrig [29] introduce a different reduction. To support NTT computations, the prime number q must have primitive $2n$-roots of unity where n is a power of two. This implies that $q - 1$ is divisible by a power of two: we have $q - 1 = k2^m$ where k is an odd number. In our case $q = 12289$ so $m = 12$ and $k = 3$. Then the Longa-Naehrig reduction of an integer x is defined as

$$\mathsf{red}(x) = k \times (x \bmod 2^m) - \lfloor x/2^m \rfloor. \tag{2}$$

This reduction can be efficiently implemented using shift and mask. Figure 3 shows two functions that implement $\mathsf{red}(x)$ and $\mathsf{red}(xy)$ for 32bit signed integers. A key property is $\mathsf{red}(x) = kx \pmod{q}$. In addition, we have useful bounds:

$$\begin{aligned} |\mathsf{red}(x)| &\le |x|/2^m + (q - k) \\ |\mathsf{red}(wx)| &\le k|x| + (q - k) \quad \text{if } 0 \le w < q \end{aligned} \tag{3}$$

By using this reduction, Longa and Naehrig propose the procedure shown[1] in Figure 4. As before, the input is an array `a` of `n` integers in standard order and the array elements are assumed to be in the range $[0, q)$. The NTT is computed in place and the result is stored in `a` in bit-reversed order. Array `p` contains precomputed constants also in the range $[0, q)$. These are powers of ω appropriately scaled to cancel the factor k introduced by the reduction (i.e., replacing $\omega^j \bmod q$

[1] We have modified the original slightly.

by $(k^{-1}\omega^j) \bmod q)$. This procedure mirrors the basic implementation in Figure 2, except for line 15. This line performs an additional reduction. It is there to prevent numerical overflows by reducing the magnitude of all elements in the array. Because of this extra reduction, the result of the procedure is not quite the NTT but a scaled version. This scaling can be corrected later by adapting the inverse NTT calculation [29].

```
 1  void ntt_red_ct_std2rev(int32_t *a, uint32_t n, const uint16_t *p) {
 2    uint32_t j, s, t, u, d;
 3    int32_t x, y, w;
 4
 5    d = n;
 6    for (t = 1; t < n; t <<= 1) {
 7      d >>= 1;
 8      u = 0;
 9      for (j = 0; j < t; j++) {
10        w = p[t + j]; // w_t^bitrev(j)
11        u += 2 * d;
12        for (s = u; s < u + d; s++) {
13          y = a[s];
14          x = mul_red(a[s + d], w);
15          if (t == 128) { y = red(y); x = red(x); }
16          a[s] = y + x;
17          a[s + d] = y - x;
18        }
19      }
20    }
21  }
```

Fig. 4. Example NTT procedure that uses the Longa-Naehrig reduction.

We are interested in verifying NTT procedures that use the Longa-Naehrig reductions. In particular, we revise the procedure of Figure 4 to avoid the extra reduction of line 15. To do this, we replace the constant coefficients stored in array `p`, by equivalent coefficients that are smaller in absolute value. We just allow these coefficients to be negative, so that they are all in the interval $[-(q-1)/2, (q-1)/2]$ instead of $[0, q-1]$. This ensures that the elements of `a` do not grow as fast during execution. Our revised procedure is the same as in Figure 4, except that elements of `p` have type `int16_t` and that line 15 is removed.

We have implemented this procedure and several variants that all use the Longa-Naehrig reduction. The example in Figure 4 takes an input array in standard order and produces an output in bit-reversed order. Variants take input in bit-reversed order and produce output in standard order. Other variants use the Gentleman-Sande method instead of the Cooley-Tukey method, and some combine NTT/INTT and multiplication by powers of ψ. All these examples are available in our software repository hosted on GitHub [15].

A critical issue is showing that no numerical overflows occur in these procedures. One can try to estimate how large the coefficients grow by using inequalities such as (3), but this is tedious and error-prone and it is difficult to get sufficiently precise bounds. Instead, we explore the use of static analysis tools.

3 Verification

To show that the NTT does not overflow, we first make assumptions on its input. As shown in Figure 5, we use an approach common to software model checkers: elements of array `a` are non-deterministic values (obtained by calling external function `int32_nd`). These values are then constrained to be in the range $[0, 12289)$ by using an "assume" statement. All the tools that we have tried support this approach.

```
#define Q 12289
extern int32_t int32_nd(void);
int main(void) {
  int32_t nd_a[16];
  int i;

  for(i = 0; i < 16; i++) {
    int32_t x = int32_nd();
    assume(x >= 0 && x < Q);
    nd_a[i] = x;
  }

  // call NTT procedure using Longa-Naehrig reduction on nd_a
  return 0;
}
```

Fig. 5. Test harness

We also need to annotate the NTT procedure with assertions to prove the absence of integer overflows. Some verification tools do this automatically. Alternatively, we can add explicit assertions as shown in Figure 6. The bounds on z in the assertion are calculated to ensure that $\texttt{red}(z)$ fits in a signed 32bit integer. In the general case, where $q - 1 = k2^m$, these bounds are as follows:

$$-2^{31+m} + 2^m(q - k) \leq z \leq 2^{31+m} + 2^m - 1.$$

```
static int32_t mul_red(int32_t x, int32_t y) {
  int64_t z;
  z = (int64_t) x * y;
  assert(-8796042698752 <= z && z <= 8796093026303);
  x = z & 4095;
  y = z >> 12;
  return 3 * x - y;
}
```

Fig. 6. Property

3.1 Out-Of-The-Box Verification Techniques

We have attempted to prove the assertion at line 4 of Figure 6 using state-of-the-art software-verification techniques and tools:

- Bounded model-checking: CBMC [6],
- Symbolic execution: SAW/Crucible [17].
- Infinite-state model checking: CPAChecker [3] and SeaHorn [24] with IC3-PDR [27].
- Abstract interpretation: SeaHorn with Abstract Interpretation.

With default backend solvers, CBMC and Crucible work on a scaled-down version of the problem with $n = 16$, but they fail when $n = 1024$ (we stopped them after more than 24 hours of computation). The infinite-state model checkers all timeout without finding an adequate inductive invariant (these tools use a default timeout of 900 s). The SeaHorn abstract interpreter finishes within seconds but cannot prove the property.

These results are not too surprising because the Longa-Naehrig reduction involves a mixture of logical and arithmetic operations that is not easy for general-purpose tools to reason about. Moreover, these computations are stored in an array and involve complex indexing that makes things even harder. For this reason, we do not believe that increasing the timeout would help the infinite-state model checkers.

CBMC and Crucible use bit-precise reasoning, which is adequate for the Longa-Naehrig reduction, but the SAT or SMT problems they generate become very hard when we increase n to 1024. With CBMC, we have managed to verify an NTT transform with $n = 1024$ but that takes several hours of computation. To perform this proof, we used the CaDiCaL SAT solver[2] instead of the default[3]. In this verification, CBMC produces a problem with more than five million Boolean variables and 25 million clauses. CaDiCal 1.2.1 can show that this problem is unsatisfiable (and thus that our assertions have no counterexamples) in 9026 s. CBMC needs 287 s to generate the SAT instance.

3.2 Proofs by Abstract Interpretation

A more scalable solution is to devise specialized techniques based on abstract interpretation. Since we have to compute safe bounds on the value of array elements, we choose the interval abstract domain [8] as the numerical domain. However, the bitwise operations used by the Longa-Naehrig reduction cause difficulties for this domain. We solve this problem by defining a custom transfer function[4] to model the effect of lines 10-12 in Figure 3.

[2] http://fmv.jku.at/cadical/

[3] By default, CBMC relies on MiniSAT 2.2.1, but CBMC can convert bounded model checking problems to the DIMACS format used by all modern Boolean satisfiability solvers.

[4] An interval domain over machine arithmetic such as e.g., [33] can reason more precisely about bitwise operations. However, the advantage of having a specialized transfer function is that it can be modeled in other verification tools.

Specialized Transfer Function The interval domain abstracts the set of possible values of a variable as an interval. The abstract values are either non-empty intervals with finite or infinite bounds, or a special symbol $\bot$ denoting error or unreachable:

$$\mathcal{I} = \{[a, b] \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}, a \leq b\} \cup \{\bot\}$$

The greatest element is $[-\infty, +\infty]$ and the least element is $\bot$. The concretization function $\gamma_{\mathcal{I}}$ is defined in the natural way: $\gamma_{\mathcal{I}}(\bot) = \emptyset$ and $\gamma_{\mathcal{I}}([a, b]) = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$. The ordering between intervals is defined as $[a, b] \sqsubseteq_{\mathcal{I}} [c, d] \Leftrightarrow (a \geq c) \wedge (b \leq d)$. The least upper bound of two intervals is defined as $[a, b] \sqcup_{\mathcal{I}} [c, d] = [\min(a, c), \max(b, d)]$. The transfer functions for addition ($+_{\mathcal{I}}$) and subtraction ($-_{\mathcal{I}}$) are defined as $[a, b] +_{\mathcal{I}} [c, d] = [a + c, b + d]$ and $[a, b] -_{\mathcal{I}} [c, d] = [a - d, b - c]$, respectively.

The transfer function for the Longa-Naehrig reduction, red, is defined as follows for finite intervals:

$$\mathsf{red}_{\mathcal{I}}([a, b]) = \Big[\mathsf{red}(\max(b \,\&\, {\sim}4095, a)), \mathsf{red}(\min(a \mid 4095, b))\Big] \tag{4}$$

The operators & and | are the usual bitwise *and* and bitwise *or*, and ~ is bitwise *negation*. Generalizing to infinite bounds is straightforward. We denote the interval domain extended with $\mathtt{red}_{\mathcal{I}}$ as $\mathcal{I}_{red}$.

Lemma 1 ($\mathsf{red}_{\mathcal{I}}$ is sound). $\{\mathsf{red}(x) \mid x \in \gamma_{\mathcal{I}}([a, b])\} \subseteq \mathsf{red}_{\mathcal{I}}([a, b])$

Proof: If we write $x = 4096d + r$ where $0 \leq r < 4096$ (by Euclidean division), then we have $\mathsf{red}(x) = 3r - d$. If $a \leq x \leq b$, the smallest value of $\mathsf{red}(x)$ is obtained by setting d as large as possible and r as small as possible. Let x_0 denote the largest integer such that $x_0 \leq b$ and $x_0 \bmod 4096 = 0$, then x_0 is equal to $b \,\&\, {\sim}4095$. Either $a \leq x_0$, in which case the minimum of red in $[a, b]$ is reached at x_0, or $a > x_0$, in which case the minimum is reached at a. Similarly, to find the maximum of red in $[a, b]$ we must make d as small as possible and r as large as possible. Let $x_1 = a \mid 4095$ then x_1 is the smallest integer such that $x_1 \geq a$ and $x_1 \bmod 4096 = 4095$. The maximum of red is reached either at point x_1 if $x_1 \leq b$ or at b otherwise.

Arrays and Loops Abstract interpretation tools (such as our SeaHorn analyzer [24]) use abstract domains to represent arrays and compute fixed points for loops using techniques such as widening. By default, SeaHorn uses a simple array domain $\mathcal{A}(\mathcal{D})$ [4] that models each memory region offset separately with a *synthetic* variable in the underlying numerical domain $\mathcal{D}$. The synthetic variables are *smashed* into a single *summary* variable if an array write occurs at an index that cannot be determined constant during analysis. Once an array is smashed, all the array writes are modeled as weak updates.

Such an array domain is not precise enough for our problem. To understand why, let us focus on `ntt_ct_std2rev` in Figure 2. The abstract array representing `a` gets first "smashed" because it is accessed at non-constant indices. After one loop iteration, all elements of `a` are then represented by a summary variable a_{sum}. Lines 16-17 perform operations: `a[s + d] = a[s] - x; a[s] = a[s] + x;`, which make a_{sum} both increase and decrease by `x`. Eventually, widening must be applied, which loses the lower and upper bounds of a_{sum} (i.e., the final abstraction is $[-\infty, +\infty]$).

More precise array abstractions [16,22,25,10] together with a more precise widening strategy (e.g., widening with thresholds [28]) could potentially help. A simpler approach is loop unrolling. Once we fix n, all the loops and arrays in our NTT examples are statically bounded. Modulo scalability issues, this means that all loops can be fully unrolled. After unrolling, the loss of precision due to the array smashing abstraction and the widening operator disappear and, therefore, $\mathcal{A}(\mathcal{I}_{red})$ can prove that the assertion holds.

SeaHorn Results We have modified SeaHorn to support the analysis method just described. The SeaHorn Verification Framework extends the LLVM compiler infrastructure with verification techniques based on Software Model Checking and Abstract Interpretation. The SeaHorn abstract interpreter is called Crab [12]. Crab does not analyze directly LLVM bitcode but instead, it analyzes a control-flow graph (CFG) language[5] from which equation systems are extracted. These equations are solved using a chaotic iteration strategy [2] based on Bourdoncle's weak topological ordering. Crab implements general-purpose numerical domains such as intervals [8], congruences [23], zones [30], octagons [31], and polyhedra [11]. Crab also implements combination methods such as direct and reduced products [9].

We have added the transfer function for red to Crab's interval domain and we leverage LLVM to fully unroll all the loops. On the resulting loop-free code, Crab with domain $\mathcal{A}(\mathcal{I}_{red})$ can prove that all the assertions hold. Once loops are unrolled, we can also employ an existing LLVM-optimization known as Scalar Replacement of Aggregates (SROA) to further improve scalability. In our examples, SROA combined with loop-unrolling and function inlining results in LLVM code where every array element `a[i]` is treated as a single scalar variable. After this optimization, all reasoning is performed in the interval domain $\mathcal{I}_{red}$ and there is no abstract array domain anymore.

Experimental results are shown in Tables 1 and 2. Table 1 shows verification time when LLVM optimizations are not used. Table 2 shows results after aggressive LLVM optimizations including SROA. All experiments were carried out on a 2.6GHz 6-Core Intel Core i7 MacOS laptop with 32GB of memory. The examples in the table are variant implementations of the forward and inverse NTT using both the Cooley-Tukey and the Gentleman-Sande variants. In most tests, we used a fixed value for ψ and ω (i.e., $\psi = 1014$ and $\omega = 8209$). In such cases, the array `p` contains explicit constants. In examples ntt_red1024c and ntt_red1024d, we treat array `p` symbolically. We initialize it with non-deterministic values in the range $[-6144, 6144]$ (i.e., $[-q/2, +q/2]$). The results show then that our NTT

[5] Translation from LLVM bitcode to Crab CFG is implemented by a SeaHorn component called Clam [5].

Program	Description	Num Checks	Time (sec)
intt_red1024	inv CT/std2rev, $\psi = 1014$	2026	900
intt_red1024b	inv CT/rev2std, $\psi = 1014$	2026	972
ntt_red1024	CT/std2rev, $\psi = 1014$	2026	923
ntt_red1024b	CT/rev2std, $\psi = 1014$	2026	836
ntt_red1024c	CT/std2rev	1974	1151
ntt_red1024d	CT/rev2std	1974	1258
ntt_red1024e	GS/std2rev, $\psi = 1014$	8194	8265
ntt_red1024f	GS/rev2std, $\psi = 1014$	8194	8115

Table 1. SeaHorn with Crab with $\mathcal{A}(\mathcal{I}_{red})$ on NTT procedures. All programs are safe (no integer overflow is possible). CT means Cooley-Tukey. GS means Gentleman-Sande.

Program	Description	Num Checks	Time (sec)
intt_red1024	inv CT/std2rev, $\psi = 1014$	8188	9
intt_red1024b	inv CT/rev2std, $\psi = 1014$	8188	9
ntt_red1024	CT/std2rev, $\psi = 1014$	8188	9
ntt_red1024b	CT/rev2std, $\psi = 1014$	8188	9
ntt_red1024c	CT/std2rev	8194	9
ntt_red1024d	CT/rev2std	8194	11
ntt_red1024e	GS/std2rev, $\psi = 1014$	8188	10
ntt_red1024f	GS/rev2std, $\psi = 1014$	8188	10

Table 2. SeaHorn with Crab with $\mathcal{I}_{red}$ on NTT procedures aggressively optimized through inlining followed by LLVM Scalar Replacement of Aggregates (SROA). All programs are safe (no integer overflow is possible). CT means Cooley-Tukey. GS means Gentleman-Sande.

procedures do not suffer integer overflows as long as all elements of `p` are in this range, which implies no overflow for any choice of ω.

Without LLVM optimizations, SeaHorn can prove all properties with runtimes of the order of tens of minutes to a few hours. The LLVM optimization pass has a dramatic impact on performance, reducing the runtime to seconds. For comparison, we managed to prove the first example in both tables with CBMC and CaDiCaL but the verification took more than two and a half hours of CPU time.

Verification by Source-Code Transformation We now examine an alternative approach that does not build on a specialized tool such as SeaHorn. Instead, we can perform abstract interpretation by transforming the source code to operate in the abstract domain. This idea is illustrated in Figure 7. The figure shows an NTT procedure converted to work in the abstract domain (i.e., intervals) rather than in the concrete domain (i.e., 32bit integers).

We implement the interval domain as sketched in the figure, namely, we represent an interval by a pair of 64bit signed integers. The 64bit bounds are sufficient for our application but more general representations could be considered (e.g., infinite bounds or arbitrary-precision integers). We implement the usual transfer functions for arithmetic operators such as, the functions `add` and `sub` used in Figure 7. We also add specialized transfer function to handle the Longa-Naehrig reduction. For example, function `red_scale` in the figure is the transfer for the operation `mul_red(x, y)` of Figure 3 in the case where one argument is a constant and the other is an interval. In other words, `red_scale(`w`, `a`)` computes an interval $[l, h]$ such that $\forall x : l_a \leq x \leq h_a \Rightarrow l \leq \mathsf{red}(wx) \leq h$, where w is a scalar constant and a is the interval $[l_a, h_a]$.

As shown in Figure 7, input to the abstract NTT function is now in the abstract domain and consists of an array `a` of `n` intervals. We also instrument the abstract function with code to print the abstract interpretation results at every main iteration and to check that all the intervals are included in $[-2^{31}, 2^{31} - 1]$ (which implies that all intermediate results fit in signed 32bit integers). To show that no integer overflow is possible, we just execute the abstract procedure on a suitable array of input intervals such as follows:

```
interval_t *a[1024];
for (int i=0; i<1024; i++) {
  a[i] = interval(0, Q-1);
}
abstract_ntt_red_ct_std2rev(a, 1024, ntt_red1024_omega_powers_rev);
```

This method is not fully general but it works in our context because all computations are bounded. We replace the concrete array `a` of 32bit integers by an array of intervals. All other variables in the procedure (i.e., loop counters, array indices, and bounds) remain concrete. Executing the abstract program computes safe bounds on the value of the concrete array element `a[i]`. We check that these bounds on `a[i]` are compatible with our concrete implementation using 32bit arithmetic.

Although the source-code transformation could be automated, we currently rewrite the code by hand. The interval domain and transfer functions are implemented as a separate library. This analysis method is simple (it requires only a C compiler) and it is very efficient and scalable. Table 3 shows verification runtimes for the same examples as before. All runtimes are now less than 1 s.

```c
// abstract domain
typedef struct interval_s {
  int64_t min;
  int64_t max;
} interval_t;

// basic operations
extern interval_t *add(const interval_t *a, const interval_t *b);
extern interval_t *sub(const interval_t *a, const interval_t *b);

// Reduction:
// 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);

// abstract version of ntt_red_ct_std2rev
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);

    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);
}
```

Fig. 7. Abstract NTT procedure. Array `a` is an array of intervals. Functions `red_scale`, `add`, and `sub` operate on intervals. Function `show_intervals` prints intermediate results and checks for overflows.

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	0.58
ntt_red1024e	GS/std2rev, $\psi = 1014$	0.21
ntt_red1024f	GS/rev2std, $\psi = 1014$	0.19

Table 3. Verification Using Source-Code Transformation

4 Discussion and Future Work

The approach we use to prove the absence of overflows in NTT procedures is based on loop unrolling and abstract interpretation. By taking advantage of the special structure of the forward and inverse NTT transform, we have developed two scalable verification methods. One is implemented in the SeaHorn analyzer and makes aggressive use of existing LLVM optimization. The other approach rewrites the source code to operate in the abstract domain and requires only a C compiler. These techniques rely on a key property, namely, all NTT computations are bounded once we fix the parameter n. A second major ingredient is the definition of special transfer functions to handle the Longa-Naehrig reduction in the interval domain.

Our most general results (examples ntt_red1024c and ntt_ref1024d) are that the Longa-Naehrig procedures we analyze are safe for $n = 1024$ as long as the constants in array p are all within the interval $[-(q-1)/2, (q-1)/2]$. This result no longer holds for $n = 2048$. However, these bounds on p[i] are not precise. The actual value of p[i] depends on i and the parameters ψ and n, and all elements of p are powers of $\omega = \psi^2 \bmod q$. For a fixed n there are n possible choices for parameter ψ. We can then exhaustively enumerate all possible values for ψ, construct the corresponding constant table p, and verify the NTT procedures for this p. Our abstract-interpretation approach is fast enough to enable this exhaustive analysis. With this method, we can show that the NTT procedures based on the Longa-Naehrig reduction are safe (for 32bit arithmetic) not just for $n = 1024$ but also for $n = 2048$.

We can also show that no arithmetic overflow occurs under weaker assumptions than discussed in this paper. In particular, we can relax the assumption that input elements in array a are between 0 and $q - 1$. For $n = 1024$, the procedures remain safe for input in larger intervals.

In our verification, we have mostly considered fully automated verification tools. In principle, other tools—such as, Frama-C [13]—that support analysis of C code by deductive methods could also be used. However, the main issue with using such tools in our applications is finding program annotations to show that the procedures are correct. This amounts to finding appropriate inductive loop invariants for the NTT procedures. We do not know automated methods for finding such invariants other than the interval abstraction we propose. Alternatives may include hand calculation using inequalities such as 3, but it is difficult to derive precise enough bounds by hand. It is also unclear whether the default SMT solvers used by Frama-C can handle the bit-shift and bit-mask operations involved in the Longa-Naehrig reduction.

All our analysis so far has focused on soundness, namely, the absence of integer overflow. We are also interested in automated methods to prove functional correctness of different NTT implementations. An avenue we would like to explore is showing that the NTT procedure is linear, which we hope can be done with automated tools. If we can prove that, then it will be enough to test the NTT on a finite set of input vectors (i.e., a basis of $\mathbb{Z}_q[X]/(X^n+1)$) to prove that it is correct everywhere.

The NTT procedures that we discussed, and the verification examples and tools are available in an open-source software repository hosted on GitHub [15]. Our verification work was motivated by our implementation of Bliss, which is also available there [14]. SeaHorn, Crab, and Clam are also open-source and also hosted on GitHub [37,12,5].

5 Conclusion

We have presented an experiment in verifying that an optimized implementation of the Number Theoretic Transform (NTT) is free of integer overflows. Although this implementation consists of few lines of code, it is very challenging for current software verification technology due to a mix of array manipulation and bitwise operations.

We have shown that combining static loop unrolling with a specialized abstract interpretation method solves the problem. The technical foundations of our work are not novel since the techniques used here are well-known. We believe that verification of NTT algorithms is a good domain to demonstrate the usefulness of verification tools. We look forward to better abstractions and algorithms to verify this kind of algorithms in a more efficient way.

Acknowledgments

Our work on the NTT benefited from many discussions with Tancrède Lepoint. The work was partially supported by NSF Grants CCF-1816936 and CCF-1817204.

References

1. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key exchange - A new hope. In: USENIX Security Symposium. pp. 327–343. USENIX Association (2016)

2. Amato, G., Scozzari, F.: Localizing widening and narrowing. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7935, pp. 25–42. Springer (2013)
3. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: CAV. pp. 184–190 (2011)
4. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: Mogensen, T., Schmidt, D., Sudborough, I.H. (eds.) The Essence of Computation: Complexity, Analysis, Transformation (2002)
5. Clam: Crab for Llvm Abstraction Manager, available at https://github.com/seahorn/crab-llvm
6. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: TACAS. pp. 168–176 (2004)
7. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Mathematics of Computation **19**(90), 297–301 (April 1965)
8. Cousot, P., Cousot, R.: Static Determination of Dynamic Properties of Programs. In: Proceedings of the second international symposium on Programming, Paris, France. pp. 106–130 (1976)
9. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL. pp. 269–282 (1979)
10. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: POPL. pp. 105–118. ACM (2011)
11. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978. pp. 84–96. ACM Press (1978)
12. CoRnucopia of ABstractions: A language-agnostic library for abstract interpretation, available at https://github.com/seahorn/crab
13. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) Intenational Conference on Software Engineering and Formal Methods (SEFM 2012). Lecture Notes in Computer Science, vol. 7504, pp. 233–247. Springer (2012)
14. BLISS Implementation: Bimodal Lattice Signature Schemes, available at https://github.com/SRI-CSL/Bliss
15. An Implementation of the Number Theoretic Transform, available at https://github.com/SRI-CSL/NTT
16. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: Beyond strong vs. weak updates. In: Gordon, A.D. (ed.) Proceedings of the 19th European Symposium on Programming. vol. 6012, pp. 246–266 (2010)
17. Dockins, R., Foltzer, A., Hendrix, J., Huffman, B., McNamme, D., Tomb, A.: Constructing semantic models of programs with the Sotware Analysis Workbench. In: Blazy, S., Chechik, M. (eds.) Verified Software, Theories, Tools, and Experiments (VSTTE 2016). Lecture Notes in Computer Science, vol. 9971, pp. 56–72. Springer (2016). https://doi.org/10.1007/978-3-319-48869-1_5
18. Ducas, L.: Accelerating Bliss: the geometry of ternary polynomials. Cryptology ePrint Archive, Report 2014/874 (2014), http://eprint.iacr.org/2014/874

19. Ducas, L., Durmus, A., Lepoint, T., Lyubashevsky, V.: Lattice signatures and bimodal Gaussians. In: Canettu, R., Garay, J.A. (eds.) Advances in Cryptology — CRYPTO 2013. Lecture Notes in Computer Science, vol. 8042, pp. 40–56 (August 2013). https://doi.org/10.1007/978-3-642-40041-4_3
20. Fog, A.: Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. www.agner.org/optimize (2020)
21. Gentleman, W.M., Sande, G.: Fast Fourier transforms—for fun and profit. In: AFIPS'66: Proceedings—Fall Joint Computer Conference. pp. 563–578 (November 1966). https://doi.org/10.1145/1464291.1464352
22. Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. In: POPL. pp. 338–350. ACM (2005)
23. Granger, P.: Static analysis of arithmetical congruences. International Journal of Computer Mathematics **30**, 165–190 (1989)
24. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 343–361 (2015)
25. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: PLDI. pp. 339–348. ACM (2008)
26. Harvey, D.: Faster arithmetic for number-theoretic transforms. Journal of Symbolic Computation **60**, 113–119 (January 2014). https://doi.org/j.jsc.2013.09.002
27. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods in System Design **48**(3), 175–205 (2016)
28. Lakhdar-Chaouch, L., Jeannet, B., Girault, A.: Widening with thresholds for programs with complex control graphs. In: Bultan, T., Hsiung, P.A. (eds.) Automated Technology for Verification and Analysis. vol. 6996, pp. 492–502 (2011)
29. Longa, P., Naehrig, M.: Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In: Foresti, S., Persiano, G. (eds.) Cryptography and Network Security (CANS 2016). Lecture Notes in Computer Science, vol. 10052, pp. 124–139 (November 2016). https://doi.org/10.1007/978-3-319-48965-0_8
30. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) Programs as Data Objects, vol. 2053, pp. 155–172 (2001)
31. Miné, A.: The Octagon abstract domain. Higher-Order and Symbolic Computation **19**(1), 31–100 (2006)
32. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation **44**(170), 519–521 (April 1985)
33. Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Signedness-agnostic program analysis: Precise integer bounds for low-level code. In: Jhala, R., Igarashi, A. (eds.) Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7705, pp. 115–130
34. Pöppelmann, T., Güneysu, T.: Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In: Hevia, A., Neven, G. (eds.) Progress in Cryptology - LATINCRYPT 2012. Lecture Notes in Computer Science, vol. 7533, pp. 139–158. Springer (October 2012). https://doi.org/10.1007/978-3-642-33481-8_8

35. Pöppelmann, T., Oder, T., Güneysu, T.: High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In: Lauter, K.E., Rodríguez-Henríquez, F. (eds.) Progress in Cryptology - LATINCRYPT 2015. Lecture Notes in Computer Science, vol. 9230, pp. 346–365. Springer (August 2015). https://doi.org/10.1007/978-3-319-22174-8_19
36. Roy, S.S., Vercauteren, F., Mentens, N., Chen, D.D., Verbauwhede, I.: Compact ring-LWE cryptoprocessor. In: Batina, L., Robshaw, M. (eds.) Cryptographic Hardware and Embedded Systems (CHES 2014). Lecture Notes in Computer Science, vol. 8731, pp. 371–391. Springer (September 2014). https://doi.org/10.1007/978-3-662-44709-3_21
37. SeaHorn verification framework, available at https://github.com/seahorn/seahorn
38. Warren, H.S.: Hacker's Delight – Second Edition. Addison-Wesley (2013)
39. Winkler, F.: Polynomial Algorithms in Computer Algebra. Texts and Monographs in Symbolic Computation, Springer (1996)