Generating Programs for Polynomial Multiplication with Correctness Assurance

Ryo Tokuda
Department of Computer Science
University of Tsukuba
Tsukuba, Japan
tokuda@logic.cs.tsukuba.ac.jp

Yukiyoshi Kameyama
Department of Computer Science
University of Tsukuba
Tsukuba, Japan
kameyama@acm.org

Abstract

Program-generation techniques prevail in domains that need high performance, such as linear algebra, image processing, and database. Yet, it is hard to generate high-performance programs with correctness assurance, and cryptography needs both. Masuda and Kameyama proposed a DSL-based framework for implementing a program generator, an analyzer, and a formula generator, and obtained an efficient and correct implementation of Number-Theoretic Transform (NTT) that is necessary for many cryptographic algorithms.

This paper advances their study in two ways. First, we develop a generation-and-analysis framework so that program generation is driven by program analysis. As a concrete result, we have found an optimization missed in previous studies. Second, we investigate whether the framework can be applied to other algorithms, including inverse NTT. By combining generated programs, we have obtained an efficient and correct implementation of polynomial multiplication, the key for several post-quantum cryptographic algorithms.

CCS Concepts: • **Software and its engineering → General programming languages**; • **Security and privacy → Cryptography**.

Keywords: Program Generation, Analysis, and Verification, Number-Theoretic Transform, Post-Quantum Cryptography

ACM Reference Format:
Ryo Tokuda and Yukiyoshi Kameyama. 2023. Generating Programs for Polynomial Multiplication with Correctness Assurance. In *Proceedings of the 2023 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation (PEPM '23), January 16–17, 2023, Boston, MA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3571786.3573017

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PEPM '23, January 16–17, 2023, Boston, MA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0011-8/23/01…$15.00
https://doi.org/10.1145/3571786.3573017

1 Introduction

Post-quantum cryptography (PQC) is a central research topic in cryptography, as the state-of-the-art cryptographic algorithms such as RSA and Elliptic Curve may be compromised by quantum computers. The National Institute of Standards and Technology (NIST) has been conducting the Post-Quantum Cryptography Standardization process, and among the four candidates that advanced to the final round, three algorithms are based on Ring Learning with Errors (RLWE) [1], which is considered as one of the most promising hardness assumptions for the post-quantum era [17]. The key ingredient of RLWE is the NTT-based multiplication of polynomials over a certain ring [16].

For any implementations of cryptographic algorithms, efficiency and correctness are must-be-satisfied things, but achieving both of them is non-trivial. Even though the underlying *algorithm* written in a scientific paper is proved correct, its implementation can be wrong, since highly efficient implementations often use low-level optimizations that are often written in an assembly language, and are correct only for a particular set of parameters or assumptions. Furthermore, more and more implementations are proposed every year, which makes it unrealistic to formalize all such implementations in proof assistants and prove their correctness by humans.

This is the point where programming-generation techniques may help. Masuda and Kameyama [18, 19] proposed an embedded-DSL-based method via the tagless-final embedding [9], and implemented the following three generators for the Cooley-Tukey algorithm for Number-Theoretic Transform (NTT), which is a Fast Fourier Transform (FFT) algorithm over the field $\mathbb{Z}_q$ (integers modulo q):

- a program generator in the standard sense, which targets OCaml programs, C programs, or C programs with SIMD instructions.
- an interval generator, which works as a program analyzer for integer overflows for the generated program. They found that some reduction in the generated program is redundant (no overflow occurs even without the reduction), and can be eliminated, which leads to a more optimized program.
- a formula generator where the formula expresses the output of the generated program as a mathematical

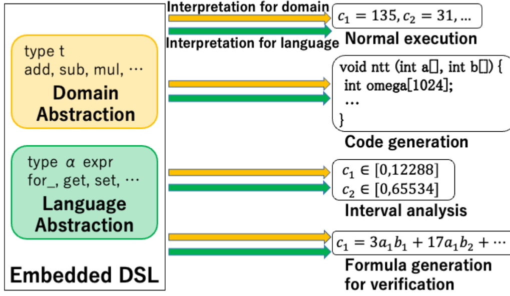

Figure 1. Tagless-Final Framework

formula (such as polynomial) over input variables. By running the formula generator for the given parameters and comparing the output formula with the definitional formula, they successfully proved the functional correctness of the generated program for the chosen parameters.

The tagless-final embedding for DSL makes it easy and handy to realize the above three (and more) generators. See Figure 1 for an illustration.

A user (a DSL programmer) of their framework only has to test the generated programs and rewrite the DSL program by inspecting the results of the computation. This quick loop allows them to find a new optimization for NTT, namely, they found some reductions are redundant by showing that the program without them is still free from integer overflow.

Although Masuda and Kameyama have applied the above framework to one particular case of the Cooley-Tukey algorithm for NTT, their work has raised several interesting research questions as follows:

- Is the program generated by their method optimal in the number of reductions?
  Since their loop contains a human, it is interesting to automate the search for correct and efficient programs.
- Does their method work only for their example program, or is it applicable to other parameters, other implementations, or other algorithms?
  Since they applied their method for a particular algorithm of NTT only, it remains to test the applicability of their method to other algorithms, implementations, and parameters.
- How can we assure that the program analyzers and formula generators are correct?
  They had a relatively big trusted computing base, since the correctness of each generator was assumed in their method. Reducing it to smaller one was left as future work.

In this paper, we answer the first two questions and discuss the third one.

To answer the first question, we have re-implemented Masuda and Kameyama's framework, which is simpler than theirs and enables us to explore more optimization opportunities. We found new optimizations in the NTT algorithm that reduce the number of reductions by 69%. This discovery is itself a new result for cryptography, however, we want to be sure that this is the most optimal. Hence, we implemented a search procedure to explore all possible combinations of inserting/deleting the reductions to find the most efficient one among the correct ones. Although our search procedure uses a simple brute-force algorithm, an interesting point is that we do *not have to generate* programs to analyze them, since our analyzer is derived from the DSL program, not from the generated program. The same holds for the formula generator; we do *not have to generate* programs to verify them. Although this is a simple observation, it has a big advantage in reducing the cost for search.

To answer the second question, we have implemented inverse NTT (INTT) and polynomial multiplication, which is a performance bottleneck of RLWE-based cryptographic algorithms. Although INTT could be implemented by the same algorithm as NTT, a different algorithm (the Gentleman-Sande algorithm) is commonly used for polynomial multiplication for efficiency reasons. We have applied the implementation framework to the Gentleman-Sande algorithm and polynomial multiplication to test the extensibility of the framework. The result is promising; our framework enables one to automatically prove that no overflow or underflow may occur during the execution of generated code, and that eliminating certain reductions from the generated code is not possible, thus it is the most optimal one in our search space. Our performance measurements have shown the effects of the above optimizations.

We also proved the functional correctness of the generated code for INTT and the overall polynomial multiplication. The method used here is essentially the same as the one by Masuda and Kameyama, but we had to extend the domain for the output formula, since the output of NTT is a linear polynomial over an input (thus we need only n integers to represent an element in the output where n is the size of the input sequence), while the output of polynomial multiplication is a quadratic polynomial over two input sequences (thus we need n^2 integers to represent an element). The extension is done without a problem, and we have successfully shown that the output formulas of polynomial multiplication coincide with those in the defining formula of the problem.

The third question is clearly difficult to answer in a single paper, as we need to formalize our generators themselves to fully address it. We will discuss the issues in this paper.

The rest of this paper is organized as follows: Section 2 gives the background of this research. We explain the optimization of the NTT and INTT algorithms and apply it to polynomial multiplication in Section 3. Section 4 shows the search process for an optimal program and provides experimental results. We verify the functional correctness of our implementation for the polynomial multiplication algorithm

in Section 5. Section 6 discusses Trusted Computing Base for this verification. We compare our work with related work in Section 7 and conclude in Section 8.

2 Background

2.1 Polynomial Multiplication and Number-Theoretic Transform

The dominant computation in RLWE-based algorithms is the multiplication of two polynomials whose coefficients are in $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ where q is a predetermined prime number called a modulus parameter. The NewHope protocol has chosen the modulus parameter $q = 12289$ [2]. Modular arithmetic is the system for arithmetic over $\mathbb{Z}_q$, and operations in modular arithmetic are called *modular operations*[1]. As is well known, polynomial multiplication with coefficients being $\mathbb{Z}_q$ can be computed in $O(n \log n)$ time by FFT, its inverse transform, and element-wise multiplication.

NTT is a discrete Fourier transform (DFT) over a finite field $\mathbb{Z}_q$. Taking $a = (a_0, a_1, \ldots, a_{n-1}) \in \mathbb{Z}_q^n$ as an input, it returns $\mathrm{NTT}_n(a) = (x_0, x_1, \ldots, x_{n-1}) \in \mathbb{Z}_q^n$ such that

$$x_i = \sum_{j=0}^{n-1} a_j \omega_n^{ij} \tag{1}$$

for $i = 0, 1, \ldots, n-1$, where ω_n is a primitive nth root of unity in $\mathbb{Z}_q$. INTT is a similar transformation to NTT and is defined by replacing ω_n with a modular multiplicative inverse of ω_n and multiplying the sum by the inverse of n. Namely, $(a_0, a_1, \ldots, a_{n-1}) = \mathrm{INTT}_n(x)$ is defined as

$$a_i = n^{-1} \sum_{j=0}^{n-1} x_j \omega_n^{-ij} \tag{2}$$

for $i = 0, 1, \ldots, n-1$.

Algorithm 1 shows the pseudocode of the Cooley-Tukey iterative algorithm [11] for NTT. It first calls the function `bit_reverse`, which reorders the input sequence a in the bit-reversed order, and stores the result in x. Then, it performs the butterfly operations in the triple loop using modular operations. We call each iteration of the outermost loop a *stage*.

2.2 Modular Reductions

Implementations in cryptographic programs are usually written in low-level programming languages such as C or assembly languages, and contain tricky low-level operations such as modular reductions for efficiency and safety against timing attacks [19]. The modular reductions used in NTT implementations are Barrett reduction [6] (`barrett_reduce`) and Montgomery reduction [21] (`montgomery_reduce`). Below we show an implementation of these reductions in C as well as that of `csub`, which can be used after the latter reduction.

[1]Note that *modular operations* are unrelated to *modules* and *modularity* in programming languages.

Algorithm 1 The Cooley-Tukey Algorithm for NTT

Input: $a = (a_0, a_1, \ldots, a_{n-1}) \in \mathbb{Z}_q^n$
Output: $(x_0, x_1, \ldots, x_{n-1}) = \mathrm{NTT}_n(a)$
  `bit_reverse`(a, x)
  for $s = 1$ **to** $\log_2 n$ **do**
    $m = 2^s$
    $o = n/m$
    for $k = 0$ **to** $n-1$ **by** m **do**
      // start of the innermost loop
      for $j = 0$ **to** $m/2 - 1$ **do**
        $\omega = \omega_n^{oj}$
        $u = x_{k+j}$
        $t = x_{k+j+m/2} \cdot \omega \mod q$
        $x_{k+j} = (u + t) \mod q$
        $x_{k+j+m/2} = (u - t) \mod q$
      end for
      // end of the innermost loop
    end for
  end for

```
uint16_t barrett_reduce(uint16_t x) {
  uint32_t u = ((uint32_t) x * 5) >> 16;
  return x - (uint16_t) (u * q);
}

uint16_t montgomery_reduce(uint32_t x) {
  uint32_t u = ((x & ((1 << 18) - 1))
               * qinv) & ((1 << 18) - 1);
  return (uint16_t) ((x + u * q) >> 18);
}

uint16_t csub(uint16_t x) {
  int16_t v = (int16_t) x - q;
  return v + ((v >> 15) & q);
}
```

The above implementations are rather tricky, and we can hardly understand their meaning without reading the explanation below.

The function `barrett_reduce` takes a 16-bit integer as an input, and returns a 14-bit integer congruent to the input modulo q. The strange constant `5` is used based on the fact that the chosen parameter q satisfies $5q < 65536 = 2^{16}$.

The function `montgomery_reduce` takes a 32-bit integer as an input, and returns it multiplied by a modular multiplicative inverse of 2^{18}. The actual output of the function is a 14-bit integer that is congruent to it modulo q. The constant `qinv` is 12287 that satisfies the equation $q \cdot \texttt{qinv} \equiv -1 \mod 2^{16}$.

The function `csub` performs conditional subtraction, which returns the difference of the input and q if it is greater than or equal to q, and returns the input otherwise.

It is now apparent that the correctness of the above implementations is far from obvious. It holds under certain

conditions on inputs and outputs, and depends on a particular choice of the parameter q (and possibly n). Verification of these implementations is strongly desired.

2.3 Embedded DSL in Tagless-Final Style

The tagless-final style [9] is a way to embed a DSL into a metalanguage. One of its features is that it separates interfaces and interpretations for DSL terms. We use the programming language OCaml as a metalanguage and its module system to realize the style. In OCaml, a signature and a module correspond to an interface and an interpretation, respectively. Types and functions that will be defined in modules are declared in signatures. DSL terms are described in a functor that takes a module of a given signature and returns another module. We can present multiple interpretations for one DSL term by applying a functor to different modules.

Masuda and Kameyama [18, 19] introduced a framework for computing NTT in the tagless-final style embedded DSL (eDSL). Their framework can generate not only highly efficient C code with SIMD instructions, but also intervals for program analysis, and formulas for program verification. Their DSL has two components: language abstraction and domain abstraction.

Language abstraction defines the typed syntax of DSL in an abstract way. The following signature `Lang` provides an interface consisting of types `'a expr` (expressions of type `'a`) and `stmt` (statements), and functions as language primitives such as `%+` for addition of array indices and `get` for retrieving the value of an array element.

```
module type Lang = sig
  type 'a expr
  type stmt
  val (%+) : int expr ->
             int expr -> int expr
  val get : 'a array expr ->
            int expr -> 'a expr
  val set : ...
  val for_ : ...
  ...
end
```

Domain abstraction expresses the domain and operations for data values. The following signature `Domain` includes an abstract type `t` which is the type of data values, and `add` for (abstract) addition of two data values.

```
module type Domain = sig
  type 'a expr
  type t
  val lift : int -> t expr
  val add : t expr ->
            t expr -> t expr
  val sub : ...
  val mul : ...
  ...
end
```

Program 1. DSL Program for Algorithm 1

```
for_ zero (m_half %- one) one (fun j ->
  let omega =
    D.lift (roots.(unlift (o %* j))) in
  let idxkj = k %+ j in
  let2
    (get x idxkj)
    (D.mul (get x (idxkj %+ m_half)) omega)
    (fun u t ->
      seq
        (set x idxkj (D.add u t))
        (set x (idxkj %+ m_half) (D.sub u t)
        ))))))
```

Our eDSL is specified by these two signatures. We can instantiate our eDSL in several different ways by combining the interpretations for `Lang` and for `Domain`. The separation of signatures is advantageous when we reuse interpretations. For example, one can fix an interpretation for `Lang` while changing one for `Domain` to obtain a different instantiation of our eDSL.

Program 1 is a DSL-program for the innermost loop of Algorithm 1. In this program, `roots.(unlift (o %* j))` corresponds to ω_n^{oj}, and `let2 e1 e2 (fun u t -> s)` corresponds to the expression `let u = e1 in let t = e2 in s`. Here we assume that the domain abstraction is interpreted by the module `D`, hence, a domain operation `add` is referred to as `D.add`.

The DSL specified by the two signatures are abstract in the sense that we can (and must) give an *interpretation* for them to actually run the program. The important merit of the tagless-final embedding is that we can give an arbitrary interpretation as long as it conforms to the interface.

For instance, to run the program in the metalanguage (OCaml in our case), we interpret (or instantiate) `Lang` by `Run`, and `Domain` by `IntModulo`, respectively, as follows:

```
module Run : Lang = struct
  type 'a expr = unit -> 'a
  type stmt = unit -> unit
  let (%+) x y () = x () + y ()
  let get a i () = Array.get (a ()) (i ())
  let set a i x () =
    Array.set (a ()) (i ()) (x ())
  let for_ low high step body () = ...
  ...
end

module IntModulo : Domain = struct
  type 'a expr = 'a Run.expr
  type t = int
  let lift x () = x
  let add x y () = (x () + y ()) mod q
  let sub x y () = (x () - y ()) mod q
  let mul x y () = ...
  ...
end
```

These interpretations are rather straightforward except that we need to use thunks such as `x ()` to manage the evaluation order.

To interpret our DSL as a C-code generator, we interpret `Lang` by `Gen`, and `Domain` by `Code`, respectively, as follows:

```
module Gen : Lang = struct
  type 'a expr = string
  type stmt = string
  let (%+) x y = ...
  let get a i = a ^ "[" ^ i ^ "]"
  let set a i x =
    a ^ "[" ^ i ^ "]␣=␣" ^ x ^ ";\n"
  let for_ low high step body = ...
  ...
end

module Code : Domain = struct
  type 'a expr = 'a Gen.expr
  type t = string
  let lift x = string_of_int x
  let add x y = x ^ "␣+␣" ^ y
  let sub x y = x ^ "␣-␣" ^ y
  let mul x y = ...
  ...
end
```

We use strings to represent C programs. We could have used MetaOCaml as well, then well-typed OCaml programs are generated.

By giving a different interpretation for the two signatures, we can instantiate an abstract NTT program to a different interpretation. For instance, by replacing the code generator with a more involved one, low-level code including Barrett reduction and Montgomery reduction can be generated. We shall instantiate the NTT program with various interpretations to generate not only code, but also intervals as the result of program analysis, and mathematical formulas to be verified. The details will be explained in later sections.

3 Analysis-Driven Optimization

Highly efficient implementations of cryptographic algorithms in the previous section rely on rather tricky low-level code, hence automatic program analysis and verification are strongly desired. Masuda and Kameyama proposed an eDSL technique to develop program analyzers and formula generators for program verification. The program analyzer works as a discriminator of good programs from bad ones, and they discovered a new optimization in their target program by running analyzers for several program variants.

In this work, we have performed the analysis-driven optimization on several programs including the Cooley-Tukey algorithm for NTT, the Gentleman-Sande algorithm for INTT, and multiplication over polynomials. Interestingly, we found that Masuda and Kameyama's implementation can be optimized further.

This section describes how we do the analysis-driven optimization for NTT and INTT, and in the next section, we show how the procedure of proving the optimality of our implementations can be systematized.

3.1 Interval Analyzer

We have implemented an interval analyzer as an interpretation of eDSL. The module `IntModuloInterval` shown below implements our abstract domain.

```
module IntModuloInterval : Domain = struct
  type 'a expr = 'a Run.expr
  type t = int * int
  let lift x () = (x,x)
  let add x y () =
    let (x1,x2) = x () in
    let (y1,y2) = y () in
    assert
      (0 <= x1 && x1 <= x2 && x2 <= 65535);
    assert
      (0 <= y1 && y1 <= y2 && y2 <= 65535);
    let (l,h) = (x1 + y1, x2 + y2) in
    assert
      (0 <= l && l <= h && h <= 65535);
    (l,h)
  let sub x y () = ...
  let mul x y () = ...
  ...
end
```

The abstract type `t` represents the underlying domain, which is `int` in the standard interpretation, but is the product type `int * int` in the above interpretation. The intuition behind this choice is that an element of this type is the pair of a lower bound and an upper bound. Then, the addition operation takes two such pairs as inputs, and returns the pair whose lower (upper, resp.) bound is the addition of the inputs' lower (upper, resp.) bounds. The assertions **`assert`** ... in the above module check whether these bounds are valid 16-bit unsigned integers. In addition to them, we can declare the intervals of the input and the output of each operation which works as pre- and post-conditions for it. Table 1 shows such conditions for each operation.

Our interval analysis is a very simple case of abstract interpretation [12], which abstracts the program execution and performs the static analysis. Interestingly, the state-of-the-art static analyzer, Frama-C, is not capable of computing sufficiently precise bounds for our purpose, and thus cannot prove that some optimized programs do not cause overflow. This is due to the use of bit-level primitives (bit-shift and bit-mask) in arithmetic operations.

Masuda and Kameyama solved the problem with a hybrid approach; for each arithmetic operation used in NTT, one can select either a standard interval analysis, or an exhaustive computation, that is, it computes the outputs for all possible inputs and takes the minimum and maximum for the outputs, producing the most precise bounds. We have

Table 1. Pre- and Post-Conditions as Constraints on Intervals

Operation	Precondition	Postcondition
$(l, h) =$ `add` (x_1, x_2) (y_1, y_2)	$0 \le x_1 \le x_2 \le 65535 \land 0 \le y_1 \le y_2 \le 65535$	$0 \le l \le h \le 65535$
$(l, h) =$ `sub` (x_1, x_2) (y_1, y_2)	$0 \le x_1 \le x_2 \le 65535 \land 0 \le y_1 \le y_2 \le 65535$	$0 \le l \le h \le 65535$
$(l, h) =$ `mul` (x_1, x_2) (y_1, y_2)	$0 \le x_1 \le x_2 \le 65535 \land 0 \le y_1 \le y_2 \le 16383$	$0 \le l \le h \le 16383$

followed them and used the hybrid approach for our version of NTT, INTT, and polynomial multiplication. Although the exhaustive computation is computationally heavy in general, it is needed only for unary arithmetic operations that have only 2^{16} values as inputs, and our hybrid analyzer worked without problems.

3.2 Optimizing the Cooley-Tukey Algorithm

We first investigate the Cooley-Tukey algorithm for NTT as in Masuda and Kameyama [19]. Algorithm 2 shows the pseudocode of the innermost loop of Algorithm 1. It involves low-level code such as Barrett and Montgomery reductions.

Algorithm 2 The Innermost Loop of the NTT Algorithm

```
for j = 0 to m/2 − 1 do
  ω = 2^18 ω_n^{oj} mod q
  u = x_{k+j}
  t = montgomery_reduce(x_{k+j+m/2} · ω)
  if s mod 2 == 0 then
    x_{k+j} = barrett_reduce(u + t)
  else
    x_{k+j} = u + t
  end if
  x_{k+j+m/2} = barrett_reduce(u + 2q − t)
end for
```

The pseudocode is the same as the Cooley-Tukey algorithm for $\mathbb{Z}_q$ elements except that it encodes $\mathbb{Z}_q$ elements as 16-bit unsigned integers, which makes it necessary to avoid integer overflow and underflow.

For instance, the second last line of the program contains an expression $u + 2q - t$, whose rationale is that we need to add a multiple of q to avoid underflow when computing $u - t$. Barrett reduction also plays an important role to avoid overflow. It returns a 14-bit unsigned integer which is congruent to its input (a 16-bit unsigned integer) modulo q.

We have thoroughly investigated Masuda and Kameyama's analysis and found that we can eliminate Barrett reductions[2] in Algorithm 2. We have come up with a way to decrease the number of Barrett reductions and performed interval analysis to prove the absence of overflow and underflow. Our way is justified as follows: As a result of our analysis, applying the function `csub` after Montgomery reduction makes the value of t less than q. This implies that we can replace the subtraction $u + 2q - t$ by $u + q - t$ without the risk of underflow. Our analysis also shows maximum return value of the function `barrett_reduce` is 16379. By the equation $16379 + 4q = 65535 = 2^{16} - 1$, we can add four values from $[0, q]$ to u without a 16-bit overflow where u is its return value. The intervals of the values t and $q - t$ are $[0, q - 1]$ and $[1, q]$, respectively. Thus, Barrett reductions need to be applied only once in four stages after the calculations $u + t$ and $u + q - t$. Given this fact, we have refined Algorithm 2 to Algorithm 3 shown below.

Algorithm 3 The Improved Innermost Loop of the NTT Algorithm

```
for j = 0 to m/2 − 1 do
  ω = 2^18 ω_n^{oj} mod q
  u = x_{k+j}
  t = csub(montgomery_reduce(x_{k+j+m/2} · ω))
  if s mod 4 == 0 then
    x_{k+j} = barrett_reduce(u + t)
    x_{k+j+m/2} = barrett_reduce(u + q − t)
  else
    x_{k+j} = u + t
    x_{k+j+m/2} = u + q − t
  end if
end for
```

Our improvement shows a drastic reduction in the total number of Barrett reductions in the NTT program. For the case of $n = 1024$, our program applies them 4×512 times (2×512 times after the addition $u+t$ and 2×512 times after the subtraction $u + q - t$) while the previous one applies 15×512 times (5×512 times after the addition $u + t$ and 10×512 times after the subtraction $u + 2q - t$). It means that we have removed 11×512 Barrett reductions. Instead, our program applies the function `csub` 10×512 times. Accordingly, we need to check the impact on the execution speed for eliminating Barrett reductions and inserting `csub`s. We will show it in practice in the next section.

As the next target, we have applied the technique to the Gentleman-Sande algorithm [20] for INTT, which significantly differs from the Cooley-Tukey algorithm. Since our interval analyzer works for any programs written in our DSL, we only have to run it for the target program without having to rewrite it. The result shows that the original implementation [2], which allows us to use 32-bit unsigned

[2] After this elimination, the values may not be smaller than q, and holding such values in the computation would be called *lazy* reductions in the literature.

integers, does not cause overflow, but we cannot improve it. As Algorithm 4 shows, we need to apply Barrett reductions once in two stages after $u + t$.

Algorithm 4 The INTT Algorithm

```
Input: x = (x_0, x_1, ..., x_{n-1}) ∈ Z_q^n
Output: (a_0, a_1, ..., a_{n-1}) = INTT_n(x)
  for s = log_2 n to 2 by −1 do
    m = 2^s
    o = n/m
    for k = 0 to n − 1 by m do
      for j = 0 to m/2 − 1 do
        ω = 2^18 ω_n^{−oj} mod q
        u = x_{k+j}
        t = x_{k+j+m/2}
        if (log_2 n − s) mod 2 == 1 then
          x_{k+j} = barrett_reduce(u + t)
        else
          x_{k+j} = u + t
        end if
        x_{k+j+m/2} = montgomery_reduce((u + 3q − t) · ω)
      end for
    end for
  end for
  for k = 0 to n − 1 by 2 do
    ω = 2^18/n
    u = x_k
    t = x_{k+1}
    x_k = montgomery_reduce((u + t) · ω)
    x_{k+1} = montgomery_reduce((u + 3q − t) · ω)
  end for
  bit_reverse(x, a)
```

3.3 Improving Vectorized Implementation

The strength of Masuda and Kameyama's work is to generate a rather efficient vectorized implementation for NTT in the C language with SIMD instructions. We have analyzed the vectorized versions of NTT and INTT, and found a new optimization that reduces the number of Barrett reductions in their work.

The challenge for vectorized implementation is that all data must be stored in vector registers of the same size, which are 16-bit integers for our case. Then, the multiplication of two 16-bit integers is problematic, and we have to use a different low-level implementation for multiplication. Masuda and Kameyama [19] proposed an implementation that works under this constraint, and used their framework for the new low-level implementation to generate an efficient vectorized program.

We have analyzed their implementation and optimized it further. While Barrett reductions must be applied once in three stages in their implementation, they need to be applied once in four stages in ours to avoid overflow and underflow. We also analyzed the vectorized INTT program (the Gentleman-Sande algorithm) to find a similar optimization, where Barrett reductions need to be applied twice in three stages. We will later explain more details about the above issues.

3.4 Polynomial Multiplication with NTT and INTT

Finally, we have investigated polynomial multiplication with NTT and INTT. RLWE-based cryptographic algorithms typically need the multiplication of polynomials over the ring $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$ [16] where the polynomial $x^n + 1$ is carefully chosen to keep the degree of polynomials within n and to allow efficient computation.

It is well known that multiplication over degree-n polynomials can be computed in $O(n \log n)$ time using NTT and INTT, and Pöppelmann et al. [23] proposed an efficient way to compute the multiplication of two elements in the ring $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$ using certain variants of NTT and INTT. See Algorithms 5 and 6 in Appendix A for these variants.

We can compute multiplication over $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$ using these algorithms. Given two polynomials $f(x) = \sum_{i=0}^{n-1} a_i x^i$ and $g(x) = \sum_{i=0}^{n-1} b_i x^i$, their product $f(x)g(x) = \sum_{i=0}^{n-1} c_i x^i$ in $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$ is computed as

$$(c_0, \ldots, c_{n-1}) = \mathrm{INTT}^*_n(\mathrm{EWM}(\mathrm{NTT}^*_n(a_0, \ldots, a_{n-1}), \mathrm{NTT}^*_n(b_0, \ldots, b_{n-1}))) \quad (3)$$

where NTT^*_n and INTT^*_n denote Algorithm 5 and Algorithm 6, respectively, and EWM performs the element-wise multiplication.

For the variants of NTT and INTT, we have applied the same method as above to obtain optimized implementations. By running the program analyzer against the variants without having to change implementations, we have confirmed that there is no danger of overflow and underflow for the optimized programs, as we have derived by our brain.

In the next section, we will show the results of our performance measurement to show the actual speedup of the improvements.

4 Searching for an Optimal Program

In Section 3, we have investigated the implementation of the Cooley-Tukey algorithm for NTT and found a new optimization opportunity that was overlooked by previous works including Masuda and Kameyama. It remains to be seen if more optimizations are possible.

In this section, we automate the process to search for an optimal program for NTT. For this particular case, we can reason about the algorithm directly and conclude that we cannot eliminate the reduction further. However, since we want to change the implementation and algorithms repeatedly to achieve better performance, human intervention in

this process should be as small as possible. We are then naturally led to implementing a search procedure which finds the correct and efficient program.

4.1 Search Procedure

Our search program does a simple brute-force search for correct programs. The following listing shows that we search for the program where `i`, `j`, and `k` specify the stages in which the reduction should be performed.

```
module M = Algorithm(Run)(IntModuloInterval)

let ntt3 () =
  for i = 1 to logn - 2 do
    for j = i + 1 to logn - 1 do
      for k = j + 1 to logn do
        try
          ...
          let _ =
            M.ntt_stages [i;j;k] ... in
          print_endline
            (sprintf "[%d,%d,%d]:␣OK" i j k)
        with Assert_failure(_) ->
          print_endline
            (sprintf "[%d,%d,%d]:␣NG" i j k)
      done
    done
  done
```

The module `IntModuloInterval` is a program analyzer, which fails during the execution if it cannot guarantee no overflow for the generated program for the given `i`, `j`, and `k`. Similarly, we searched for non-overflow programs that omit Barrett reductions at fewer number of stages than `ntt3`. The result was negative, namely, our generated program is the most optimal in the number of Barrett reductions on our search space.

Note that we do not have to call the (standard) program generator, which means that we analyze a generated program without having the program itself! This observation is probably too simple to state in a research paper for programming languages, however, it would surprise outsiders including experts in cryptography, who actually write an implementation in our eDSL. Program-generation technique indeed helps cryptography.

4.2 Performance Measurements

We conducted performance measurements to find the effect of our optimizations. To do so, we generated C programs for NTT, INTT, and polynomial multiplication in our framework, which are then compiled and executed. The computing environment for our experiment was macOS 12.5.1, Apple M1, 16 GB memory, OCaml 4.14.0, and Clang 15.0.5.

Table 2 shows the results for the non-vectorized implementations, where NTT and INTT are the algorithms of their variants, and are the components of polynomial multiplication. The figures are the average execution time for 10 trials of 10,000 runs.

Table 2. The execution time of non-vectorized implementations (sec.)

Algorithm	Implementation	$n = 512$	$n = 1024$
NTT	(A) Previous work	0.1429	0.2818
	(B) This work	0.1305	0.2439
INTT	(C) Previous work	0.1068	0.1931
Polynomial multiplication	(A) & (C)	0.3318	0.7219
	(B) & (C)	0.3068	0.6375

Table 3. The number of Barrett reductions in vectorized implementations

Algorithm	Implementation	$n = 512$	$n = 1024$
NTT	(D) Previous work	12×256	13×512
	(E) This work	4×256	4×512
INTT	(F) Previous work	8×256	9×512
	(G) This work	5×256	6×512
Polynomial multiplication	(D) & (F)	32×256	35×512
	(E) & (G)	13×256	14×512

Table 2 suggests that our improvement on eliminating Barrett reductions leads to significant speed-up for these algorithms. More precisely, our improvement makes polynomial multiplication approximately 8-13% faster than the previous work. This indicates that the cost of Barrett reductions is high in these algorithms and that the positive effect of eliminating Barrett reductions is greater than the negative effect of inserting `csubs`.

Table 3 shows the results for vectorized implementations. It shows the numbers of Barrett reductions since measuring the actual execution time with vector instructions (SIMD instructions) needs special care for fair comparison.

Table 3 shows that our optimization significantly reduces the number of Barrett reductions. The reduction is approximately 66-69% for NTT, 33-38% for INTT, and 59-60% for polynomial multiplication.

We expect that our optimization improves the execution time in the vectorized computing environment, but its measurement is left for future work.

5 Verifying Functional Correctness

Functional correctness of an algorithm or a program means that it satisfies the input-output relation with respect to a given specification. This section shows how we ensure the functional correctness of generated code for polynomial multiplication in the previous section.

5.1 Masuda and Kameyama's Verification for NTT

Masuda and Kameyama [19] proved the functional correctness of their implementation of the Cooley-Tukey algorithm with respect to the definitional formula of NTT. It is highly non-trivial, as the program generated by their method contains low-level optimized code for arithmetic operations, as well as high-level optimizations such as lazy reduction or so-called bit-reversal. Moreover, since such implementations are rapidly updated by new optimizations, the verification procedure should be fully or largely automated without requiring human intervention.

Masuda and Kameyama [19] solved this problem by the following observations:

- It is rather difficult to verify the generated program as a whole, which would force us to reason about high-level mathematical properties and low-level implementation details simultaneously. In fact, applying the state-of-the-art SMT solver to the generated program did not work. One needs a way to decompose the generated program into a high-level part and low-level parts, but it is exactly what the eDSL approach provides.
- In cryptography, generated programs are, in most cases, straight-line code, namely, they contain no loops, conditionals, or function calls. Thus, one can express the output value as an expression over input values as variables. In fact, the output values in the NTT algorithm are expressible as linear functions over input variables if we ignore low-level details. Then proving functional correctness for the high-level part boils down to coefficient-wise checking of a linear function between the output of the generated program and the definitional formula.
- The low-level part such as Barrett reduction consists of a small but complicated code which contains bit-level operations. Luckily, the state-of-the-art SMT solver has the bit-vector theory, that can verify the functional correctness of low-level functions completely automatically.

Based on these observations, they have verified a low-level generated program and high-level generated code separately, which was successful.

5.2 Verification for Polynomial Multiplication

Our goal is to prove that the output of the polynomial multiplication program using the improved implementations for NTT and INTT is equivalent to the definition of the multiplication of two polynomials over $\mathbb{Z}_q[x]/\langle x^n+1\rangle$. Given two polynomials $f(x) = \sum_{i=0}^{n-1} a_i x^i$ and $g(x) = \sum_{i=0}^{n-1} b_i x^i$, the coefficients of the product $f(x)g(x) = \sum_{i=0}^{n-1} c_i x^i$ are defined by

$$c_i \equiv \sum_{j=0}^{i} a_i b_{i-j} - \sum_{j=i+1}^{n-1} a_j b_{n+i-j} \mod q \tag{4}$$

for $i = 0, 1, \ldots, n-1$. (Note that we compute polynomials modulo $x^n + 1$, which means x^n and -1 are congruent.)

First, we verified the functional correctness of the low-level operations Barrett reduction, Montgomery reduction, and `csub`. Unlike Masuda and Kameyama's work, the verification task was accomplished by a brute-force approach, namely, we computed the results of applying each operation to all possible inputs, and checked if the results satisfy the specification. This exhaustive check was successful, which confirmed the functional correctness of all three operations. Note that verification by the SMT solver took a long time, while our brute-force method ran very efficiently.

For the functional correctness of the higher-level part, we implemented a formula generator as an interpretation of the abstract domain following Masuda and Kameyama, but one extension was needed. Our formula generator must deal with quadratic functions over inputs, while the output of their target (an NTT algorithm) can be expressed as a linear function over inputs. By inspecting our algorithm, it turned out that we only need either a constant, a linear function over inputs, or a homogeneous quadratic function over two inputs. Hence, we instantiate the abstract type in our Formula module as follows:

```
module Formula : Domain = struct
  ...
  type t =
    | Const of int
    | Linear of int array
    | Quad of int array array
  ...
end
```

For instance, the expressions `Const(357)`, `Linear([3;5;7])`, and `Quad([[3;0;0];[0;5;0];[0;0;7]])` have type `Formula.t`, and they represent the constant 357, the linear polynomial $3a_0 + 5a_1 + 7a_2$, and the quadratic polynomial $3a_0b_0 + 5a_1b_1 + 7a_2b_2$, respectively.

We then implemented each arithmetic operation over this type. (Note that the functional correctness of low-level operations has been already proved, we can safely regard each arithmetic operation as a mathematical modular operation.) For instance, the operation `add` is instantiated as follows:

```
let add x y () =
  match (x (), y ()) with
  | (x,Const(c)) when c mod q = 0 -> x
  | (Linear(a),Linear(b)) ->
     Linear(Array.init n (fun i ->
     (a.(i) + b.(i)) mod q))
  | (Quad(a),Quad(b)) ->
     Quad(Array.init n (fun i ->
```

```
      Array.init n (fun j ->
      (a.(i).(j) + b.(i).(j)) mod q)))
  ...
  | _ -> failwith "unexpected␣expression"
```

It is a naïve implementation for multiplication over formulas. The important property here is that the results of the computation of add never go beyond the type Formula.t.

Finally, we combined the above implementation into an interpretation of Domain as follows:

```
module Formula : Domain = struct
  type 'a expr = 'a Run.expr
  type t =
    | Const of int
    | Linear of int array
    | Quad of int array array
  let add x y () = ...
  let sub x y () = ...
  let mul x y () = ...
  ...
end
```

We ran the formula generator for our polynomial multiplication algorithm under the standard interpretation for language primitives such as for_, which results in n quadratic polynomials that represent each component of the output. As an example, the following formula has been obtained as the 100th output for $n = 1024$:

```
  1 a[0]b[100] + 1 a[1]b[99] + 1 a[2]b[98]
+ 1 a[3]b[97] + 1 a[4]b[96] + 1 a[5]b[95]
+ ...
+ 12288 a[1020]b[104] + 12288 a[1021]b[103]
+ 12288 a[1022]b[102] + 12288 a[1023]b[101]
```

Producing the above formula has taken approximately 379 seconds in our computing environment described in Section 4.2.

Note that 12288 is equal to -1 modulo $q = 12289$. We have compared each output formula with the definitional formula, and confirmed that they are point-wise equivalent for $n = 512$ and $n = 1024$. This result subsumes the functional correctness of our implementation of polynomial multiplication over $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$ for the above n and $q = 12289$.

6 Discussion on Trusted Computing Base

Our verification in the previous section relies on several assumptions which we trust without proofs, namely, we have a relatively big trusted computing base (TCB).

Besides the correctness of the implementation languages (including OCaml and C for our case) and the hardware, the biggest component of our TCB is that all interpretations of the eDSL semantically agree. In this section, we discuss how one can reduce the TCB.

6.1 Tagless-Final Embedding

The tagless-final embedding of a DSL is a fairy convenient way to express an embedded DSL using an abstraction mechanism in programming languages, such as module in OCaml, trait in Scala, and type class in Haskell.

A program in the DSL is expressed by abstraction, and is given various interpretations such as execution, printing, code generation, and even more. The DSL itself gives no semantics, thus we can write any interpretation as long as it respects typings. For instance, a DSL expression `a %+ b` in our eDSL can be interpreted as addition in the standard semantics, but it can be subtraction, multiplication, or any kinds of binary operations in the non-standard semantics.

This flexibility of tagless-final embedding is useful but sometimes problematic. In order for our analysis and verification to make sense, we assumed, without a proof, that generated intervals and formulas in our interpretations correctly respect the *standard* semantics of DSL. Clearly, we need a way to constrain interpretations so that they certainly agree with the standard semantics. This is what we call the coherence problem (for interpretations).

6.2 Modular Approach

The key to solve the coherence problem (at least partially) is, again, the modularity induced by the tagless final embedding. In its basic form, all interpretations must be compositional, so proving the equivalence of each component of an interpretation with the same component of the standard semantics would imply the coherence property of the interpretation. Then, we only have to consider each primitive or operation separately, and we pick up a few cases as follows:

- Addition is interpreted by the interval generator in such a way that t adds the lower bounds (or upper bounds) of its operands, and returns the pair. This would be trivially justified by the standard semantics. Addition is interpreted by the formula generator as coefficient-wise addition over polynomials. Again, it is easy to justify it.
- Multiplication is more interesting, since it is realized by primitive multiplication and Montgomery reduction, which is a highly non-trivial implementation. If its implementation used in the interval analyzer and in the code generator is implemented independently, the result of interval analysis has nothing to do with the generated code.
  We solve this problem by providing an implementation of Montgomery reduction as a tagless-final translation so that all interpreters use the same translation as an implementation. In OCaml, a tagless-final translation is realized as a functor, a function over modules, and similarly realized in other languages. For instance, we can define the functor AddMontgomery to add a specific

implementation of Montgomery reduction in a low-level domain `LLDomain`.

```
module AddMontgomery
  : functor (D:LLDomain) -> Domain
```

Then, the semantic coherence is guaranteed provided all interpretations use `AddMontgomery`, regardless of its actual implementation.

- Extensibility.
  We may need to add a low-level operation to generate efficient code. Masuda and Kameyama introduced the operations `mullo` and `mulhi` for vectorized implementations, which multiply two 16-bit integers and return the lower or upper 16 bits as the results.
  If we have to implement the interval analysis for these operations separately from the code generator, the coherence property might be lost. Instead, we implement `mullo` in terms of existing operations such as 32-bit multiplication and the modulo operation, which will be inefficient, but is sufficient for the purpose of analysis and verification. Then, an implementation of the interval analysis for them is automatically derived from that for existing operations.
  To prove the coherence for `mullo`, we only have to prove that the implementation of `mullo` respects its implementation, which is usually easier than proving the correctness of a custom-made interval analyzer.

We have so far discussed about operations over domains only. Language primitives such as `for_` need some care, as their semantics is not necessarily compositional. However, our DSL has a very limited set of standard primitives. Therefore, it would not be difficult to formally reason about the correctness of each interpretation, but it is left for future work.

7 Related Work

There are a number of related work that aims to give highly efficient and correct implementations for cryptographic algorithms. In this section, we pick up those closely related to ours.

Navas et al. [22] presented how to show the absence of 32-bit overflows on NTT. They used the SeaHorn verification framework [14] and the Crab abstract interpretation library to prove it. This method is implemented as a C program specialized for one NTT program. Compared with their study, our framework is more general in that ours can analyze any kinds of programs that are expressible in our DSL, including implementations of NTT, INTT, and polynomial multiplication, without having to implement an analyzer for each target.

Hwang et al. [15] proposed a general verification method to ensure the correctness of highly-efficient implementations of NTT-based polynomial multiplication algorithms. Notably, their method targets real assembly code with SIMD instructions, thus their Trusted Computing Base is much smaller than ours. They successfully verified the implementations proposed by three finalists in the PQC Standardization. One drawback of their approach is that a user has to insert many assertions such as "the value of register X is the same as that of Y" into the target code in order for their verification tool CryptoLine to automatically certify the target. This means that a user of their method must understand the details of internal cryptographic algorithms, that is a big burden for many users. Moreover, it will be hard to maintain the assurance if the implementation changes quickly. Similarly to our work, they decomposed a verification problem into several pieces using the `cut` rule[3]. While they have to think about how to decompose a big assembly code, our decomposition is derived by the eDSL and its interpretations. In summary, their method gives the highest level of reliability, ours gives an easy-to-verify method so that one can implement, analyze and verify quickly and repeatedly.

Erbsen et al. [13] introduced a way to implement low-level cryptographic primitives that uses the Coq proof assistant to verify the functional correctness of their implementation. Similarly, the framework Jasmin [3] also used this assistant to develop high-speed and high-assurance cryptographic software. In addition, the library HACL* [27], written in the F* programming language, verifies the implementations of cryptographic primitives. Since the cost of verification with Coq or F* is high, our framework is more suitable for rapid prototyping, such as NTT.

Amin and Rompf [4] advocated the usefulness of generative programming in not only program generation but also verification. Their work can generate C code and ACSL specifications together, and then the code is verified using the specifications. They successfully verified memory safety, overflow safety, and functional correctness. Although their work and ours are similar in methodology, the target programs are quite different. We are mainly interested in highly efficient implementations used in cryptography where low-level tricky operations play an important role.

Studies on the acceleration of NTT implementations on Field-Programmable Gate Arrays (FPGA) have been actively done [10, 24–26]. They realize low-latency NTT implementations by various hardware-level optimizations, which brings a new challenge of verification. We hope to apply our eDSL-based approach to hardware generation, however, it is left for future work.

The PQC Standardization process is already in its final stage. CRYSTALS-Kyber [5] is a finalist in the PQC Standardization [1], which gives efficient implementations for NTT, INTT, and polynomial multiplications for a different set of security parameters ($n = 256$ and $q = 3329$). The acceleration of NTT and polynomial multiplication by Kyber is an active

[3]The `cut` rule in CryptoLine is similar to the cut rule in sequence calculus.

research topic [7, 8]. We believe that our method can be used for analyzing and verifying Kyber.

8 Conclusion

We have presented an eDSL-based framework which realizes low-level implementations for NTT, INTT, and polynomial multiplication. The systematic framework enables one to automatically generate code, analyze it, and obtain a representation of the output as a mathematical formula that can be verified outside of our framework. The result of our interval analysis gave a new optimization to reduce the number of reductions, and the search procedure for the optimal program in our search space, without having to generate a huge number of programs. We have verified the end-to-end, functional correctness of our implementation for polynomial multiplication. Altogether, we have shown how DSL-based program-generation technology contributes to generating a highly efficient implementation of cryptographic algorithms with correctness assurance.

We think that the key to this success came from the modular and extensible nature of the tagless-final embedding. In fact, we invented nothing new in programming-language technology. Rather, we found a new application of existing technology.

As future work, we hope to apply the technique exploited in this paper to more implementations for different algorithms. The most notable applications are the finalists in the PQC Standardization, such as Kyber. Other important future work is to generate and certify assembly programs and FPGAs for cryptography.

Acknowledgments

We would like to thank Masahiro Masuda and the members of the Programming Logic Group for their helpful comments. Thanks also go to the anonymous reviewers for their careful reading and constructive criticism. The authors are supported in part by JSPS Grant-in-Aid (B) 22H03563.

A The NTT and INTT Algorithms for Polynomial Multiplication

Algorithms 5 and 6 show the pseudocode of variants of NTT and INTT, which are used for polynomial multiplication. These are based on the Cooley-Tukey algorithm and the Gentleman-Sande algorithm, respectively, but they do not execute bit reversal. They use components of Ψ and Ψ^* as a factor ω instead of the powers of the primitive nth root ω_n. The arrays Ψ and Ψ^* contain powers of a square root ψ of ω_n in bit-reversed order and those of a modular multiplicative inverse of ψ in bit-reversed order, respectively.

Algorithm 5 A Variant of the NTT Algorithm

Input: $a = (a_0, a_1, \ldots, a_{n-1}) \in \mathbb{Z}_q^n$
  for $s = 1$ **to** $\log_2 n$ **do**
    $m = 2^s$
    $o = n/m$
    for $k = 0$ **to** $m/2 - 1$ **do**
      $j_1 = 2 \cdot k \cdot o$
      $j_2 = j_1 + o - 1$
      $\omega = \Psi_{m/2+k}$
      for $j = j_1$ **to** j_2 **do**
        $u = a_j$
        $t = a_{j+o} \cdot \omega \mod q$
        $a_j = (u + t) \mod q$
        $a_{j+o} = (u - t) \mod q$
      end for
    end for
  end for

Algorithm 6 A Variant of the INTT Algorithm

Input: $x = (x_0, x_1, \ldots, x_{n-1}) \in \mathbb{Z}_q^n$
  for $s = \log_2 n$ **to** 2 **by** -1 **do**
    $m = 2^s$
    $o = n/m$
    for $k = 0$ **to** $m/2 - 1$ **do**
      $j_1 = 2 \cdot k \cdot o$
      $j_2 = j_1 + o - 1$
      $\omega = \Psi^*_{m/2+k}$
      for $j = j_1$ **to** j_2 **do**
        $u = x_j$
        $t = x_{j+o}$
        $x_j = (u + t) \mod q$
        $x_{j+o} = (u - t) \cdot \omega \mod q$
      end for
    end for
  end for
  $\omega = n^{-1} \cdot \Psi^*_1 \mod q$
  for $j = 0$ **to** $n/2 - 1$ **do**
    $u = x_j$
    $t = x_{j+n/2}$
    $x_j = (u + t) \cdot n^{-1} \mod q$
    $x_{j+n/2} = (u - t) \cdot \omega \mod q$
  end for

References

[1] Gorjan Alagic, David Cooper, Quynh Dang, Thinh Dang, John M. Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl A. Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Daniel Apon. 2022. Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process. https://doi.org/10.6028/NIST.IR.8413

[2] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. 2016. Post-quantum Key Exchange - A New Hope. In *25th USENIX*

Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 327–343. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1807–1823. https://doi.org/10.1145/3133956.3134078

[4] Nada Amin and Tiark Rompf. 2017. LMS-Verify: abstraction without regret for verified systems programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 859–873. https://doi.org/10.1145/3009837.3009867

[5] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2021. CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation (version 3.02). https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf

[6] Paul Barrett. 1986. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings (Lecture Notes in Computer Science, Vol. 263)*, Andrew M. Odlyzko (Ed.). Springer, 311–323. https://doi.org/10.1007/3-540-47721-7_24

[7] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. 2022. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022, 1 (2022), 221–244. https://doi.org/10.46586/tches.v2022.i1.221-244

[8] Mojtaba Bisheh-Niasar, Reza Azarderakhsh, and Mehran Mozaffari Kermani. 2021. High-Speed NTT-based Polynomial Multiplication Accelerator for CRYSTALS-Kyber Post-Quantum Cryptography. *IACR Cryptol. ePrint Arch.* (2021), 563. https://eprint.iacr.org/2021/563

[9] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (2009), 509–543. https://doi.org/10.1017/S0956796809007205

[10] Donald Donglong Chen, Nele Mentens, Frederik Vercauteren, Sujoy Sinha Roy, Ray C. C. Cheung, Derek Chi-Wai Pao, and Ingrid Verbauwhede. 2015. High-Speed Polynomial Multiplication Architecture for Ring-LWE and SHE Cryptosystems. *IEEE Trans. Circuits Syst. I Regul. Pap.* 62-I, 1 (2015), 157–166. https://doi.org/10.1109/TCSI.2014.2350431

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press. http://mitpress.mit.edu/books/introduction-algorithms

[12] Patrick Cousot and Radhia Cousot. 2010. A gentle introduction to formal verification of computer systems by abstract interpretation. In *Logics and Languages for Reliability and Security*, Javier Esparza, Bernd Spanfelner, and Orna Grumberg (Eds.). NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 25. IOS Press, 1–29. https://doi.org/10.3233/978-1-60750-100-8-1

[13] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2020. Simple High-Level Code For Cryptographic Arithmetic: With Proofs, Without Compromises. *ACM SIGOPS Oper. Syst. Rev.* 54, 1 (2020), 23–30. https://doi.org/10.1145/3421473.3421477

[14] Arie Gurfinkel, Temesghen Kahsai, and Jorge A. Navas. 2015. SeaHorn: A Framework for Verifying C Programs (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9035)*, Christel Baier and Cesare Tinelli (Eds.). Springer, 447–450. https://doi.org/10.1007/978-3-662-46681-0_41

[15] Vincent Hwang, Jiaxiang Liu, Gregor Seiler, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2022. Verified NTT Multiplications for NISTPQC KEM Lattice Finalists: Kyber, SABER, and NTRU. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022, 4 (2022), 718–750. https://doi.org/10.46586/tches.v2022.i4.718-750

[16] Patrick Longa and Michael Naehrig. 2016. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In *Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10052)*, Sara Foresti and Giuseppe Persiano (Eds.). 124–139. https://doi.org/10.1007/978-3-319-48965-0_8

[17] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2013. On Ideal Lattices and Learning with Errors over Rings. *J. ACM* 60, 6 (2013), 43:1–43:35. https://doi.org/10.1145/2535925

[18] Masahiro Masuda and Yukiyoshi Kameyama. 2021. FFT Program Generation for Ring LWE-Based Cryptography. In *Advances in Information and Computer Security - 16th International Workshop on Security, IWSEC 2021, Virtual Event, September 8-10, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12835)*, Toru Nakanishi and Ryo Nojima (Eds.). Springer, 151–171. https://doi.org/10.1007/978-3-030-85987-9_9

[19] Masahiro Masuda and Yukiyoshi Kameyama. 2022. Unified Program Generation and Verification: A Case Study on Number-Theoretic Transform. In *Functional and Logic Programming - 16th International Symposium, FLOPS 2022, Kyoto, Japan, May 10-12, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13215)*, Michael Hanus and Atsushi Igarashi (Eds.). Springer, 133–151. https://doi.org/10.1007/978-3-030-99461-7_8

[20] Kevin Millar, Marcin Lukowiak, and Stanislaw P. Radziszowski. 2019. Design of a Flexible Schönhage-Strassen FFT Polynomial Multiplier with High- Level Synthesis to Accelerate HE in the Cloud. In *2019 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2019, Cancun, Mexico, December 9-11, 2019*, David Andrews, René Cumplido, Claudia Feregrino, and Marco Platzner (Eds.). IEEE, 1–5. https://doi.org/10.1109/ReConFig48160.2019.8994790

[21] Peter L. Montgomery. 1985. Modular Multiplication Without Trial Division. *Math. Comp.* 44 (1985), 519–521. https://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf

[22] Jorge A. Navas, Bruno Dutertre, and Ian A. Mason. 2020. Verification of an Optimized NTT Algorithm. In *Software Verification - 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20-21, 2020, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12549)*, Maria Christakis, Nadia Polikarpova, Parasara Sridhar Duggirala, and Peter Schrammel (Eds.). Springer, 144–160. https://doi.org/10.1007/978-3-030-63618-0_9

[23] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. 2015. High-Performance Ideal Lattice-Based Cryptography on 8-Bit ATxmega Microcontrollers. In *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9230)*, Kristin E. Lauter and Francisco Rodríguez-Henríquez (Eds.). Springer, 346–365. https://doi.org/10.1007/978-3-319-22174-8_19

[24] Yang Yang, Sanmukh R. Kuppannagari, Rajgopal Kannan, and Viktor K. Prasanna. 2022. NTTGen: a framework for generating low latency NTT implementations on FPGA. In *CF '22: 19th ACM International Conference on Computing Frontiers, Turin, Italy, May 17 - 22, 2022*, Luca Sterpone, Andrea Bartolini, and Anastasiia Butko (Eds.). ACM, 30–39. https://doi.org/10.1145/3528416.3530225

[25] Tian Ye, Yang Yang, Sanmukh R. Kuppannagari, Rajgopal Kannan, and Viktor K. Prasanna. 2021. FPGA Acceleration of Number Theoretic Transform. In *High Performance Computing - 36th International Conference, ISC High Performance 2021, Virtual Event, June 24 - July 2, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12728)*, Bradford L. Chamberlain, Ana Lucia Varbanescu, Hatem Ltaief, and Piotr Luszczek (Eds.). Springer, 98–117. https://doi.org/10.1007/978-3-030-78713-4_6

[26] Neng Zhang, Bohan Yang, Chen Chen, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2020. Highly Efficient Architecture of NewHope-NIST on FPGA using Low-Complexity NTT/INTT. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020, 2 (2020), 49–72. https://doi.org/10.13154/tches.v2020.i2.49-72

[27] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1789–1806. https://doi.org/10.1145/3133956.3134043

Received 2022-10-18; accepted 2022-11-15