Unified Program Generation and Verification: A Case Study on Number-Theoretic Transform

Masahiro Masuda and Yukiyoshi Kameyama

University of Tsukuba, Tsukuba, Japan
masa@logic.cs.tsukuba.ac.jp, kameyama@acm.org

Abstract. Giving correctness assurance to the generated code in the context of generative programming is a poorly explored problem. Such assurance is particularly desired for applications where correctness of the optimized code is far from obvious, such as cryptography.
This work presents a unified approach to program generation and verification, and applies it to an implementation of Number-Theoretic Transform, a key building block in lattice-based cryptography. Our strategy for verification is based on problem decomposition: While we found that an attempt to prove functional correctness of the whole program all at once is intractable, low-level components in the optimized program and its high-level algorithm structure can be separately verified using procedures of appropriate levels of abstraction.
We demonstrate that such a decomposition and subsequent verification of each component are naturally realized in a program-generation approach based on the tagless-final style, leading to an end-to-end functional correctness verification of a highly optimized program.

1 Introduction

State-of-the-art multi-stage programming languages and systems can generate highly performant code [14,15,24,25]. In terms of reliability, however, assuring correctness beyond type safety of generated code has been rarely provided and thus it remains a relatively unexplored problem. For applications where code correctness is as important as performance, this is an undesirable situation.

Cryptography is an example of such application domains. Expert cryptographers still write performance-critical code in assembly. Assembly code makes it hard to be confident in the correctness of its implementation, as well as complicates the development and maintenance process. Although there has been remarkable progress on verifying and generating code for low-level cryptographic primitives that are used today [11,26], doing full-scale verification of bleeding-edge primitives that are still being developed is costly and unrealistic. Number-Theoretic Transform (NTT) is one example of recent primitives that is of increasing interest in lattice-based cryptography. More specifically, NTT is a variant of Fast Fourier Transform specialized to a finite field; It is used to accelerate polynomial multiplication on prime-field coefficients, which is at the heart of cryptographic constructions based on the Ring learning with errors (RLWE) problem [18]. Since the RLWE problem is widely recognized as a promising hardness assumption for post-quantum cryptography, many cryptographic schemes based on the hardness of the RLWE problem have been developed, along with optimized assembly implementations of NTT [13,3,17,23]. We believe that the programming language community should be able to help implement correct and efficient code for such state-of-the-art primitives.

This work contributes a DSL-based approach to an NTT implementation, which uniformly represents code generation and verification in a single framework. Our approach is based on module-based abstraction techniques for embedded DSL implementations that are well-known in the functional programming community: Specifically, the tagless-final style [9] is used for code generation. Our framework extends our previous work on code generation [19] to accommodate program verification as an instance of interpretations of a DSL program. By exploiting the highly parameterized nature of the framework, we can realize interval analysis and symbolic computation at the level of an abstract DSL, while taking into account low-level details that are present in the generated code.

We have performed both safety property and functional correctness verification, which led us to several interesting findings. First, we found that a DSL framework based on the tagless-final style naturally enables a custom implementation of interval analysis, and that it can yield more precise bounds than those estimated by the state-of-the-art static analyzer for C programs, Frama-C value analysis tool [7], applied to the generated code. Moreover, the more precise bounds allowed us to discover a new optimization opportunity that was not known before. Second, we found that decoupling the low-level details of modular reductions from the high-level structure of the NTT algorithm is the key to carrying out the end-to-end equivalence checking against the DFT reference. We summarize our contributions as follows[1].

- A unified treatment of code generation and verification in a single framework
- Interval analysis for NTT that verifies the absence of integer overflow (Sections 4.1 and 4.2)
- A verified derivation of a new code optimization based on interval analysis (Section 4.3)
- End-to-end verification of functional correctness of the highly optimized NTT code against a textbook DFT algorithm (Section 5)

The rest of the paper is organized as follows: Section 2 gives background to this work. We describe our verification tasks concretely in Section 3 before we go into our technical contributions in Sections 4 and 5. We recap the pros and cons of our approach in Section 6. Section 7 discusses related work and we conclude in Section 8.

[1] Our code is available in https://github.com/masahi/nttverify.

2 Background

2.1 Number-Theoretic Transform

NTT is an $O(n \log n)$ time algorithm to compute Discrete Fourier Transform (DFT) on a finite field. DFT is defined as follows: Given an input $a = (a_0, a_1, ..., a_{n-1})$ such that $a_i \in \mathbb{Z}_q$, the finite field of integers modulo q, it computes $y = (y_0, y_1, ..., y_{n-1}), y_i \in \mathbb{Z}_q$ by the following formula [10]:

$$y_k = \sum_{j=0}^{n-1} a_j \omega_n^{kj} \tag{1}$$

Here, ω_n is the nth primitive root of unity modulo q, satisfying $\omega_n^n \equiv 1 \pmod{q}$. All addition and multiplication are done in modulo q. As an example of the choice of parameters, the NTT implementation in NewHope [3], which our previous work on code generation is based on, uses $n = 1024$ and $q = 12289$.

Algorithm 1 shows the pseudocode of a textbook NTT algorithm. It uses the standard Cooley-Tukey algorithm [10] and all powers of ω_n, called twiddle factors, are precomputed and stored in an array Ω. Each iteration of the outermost loop is often called a *stage*.

Algorithm 1 The pseudocode for the iterative, in-place NTT

```
1:  procedure NTT
    Input: $a = (a_0, a_1, ..., a_{n-1}) \in \mathbb{Z}_q^n$, precomputed constants table $\Omega \in \mathbb{Z}_q^n$
2:  Output: $y = \texttt{DFT}(a)$, in standard order
3:      bit-reverse($a$)
4:      for ($s = 1$; $s \le \log_2(n)$; $s = s + 1$) do
5:          $m = 2^s$
6:          $o = 2^{s-1} - 1$
7:          for ($k = 0$; $k < m$; $k = k + m$) do
8:              for ($j = 0$; $j < m/2$; $j = j + 1$) do
9:                  $u = a[k + j]$
10:                 $t = (a[k + j + m/2] \cdot \Omega[o + j]) \bmod q$
11:                 $a[k + j] = (u + t) \bmod q$
12:                 $a[k + j + m/2] = (u - t) \bmod q$
13:             end for
14:         end for
15:     end for
16: end procedure
```

The innermost loop performs the Cooly-Tukey butterfly operation with modular arithmetic. Existing work [3,23] and our code-generation framework use specialized algorithms for modular reductions. We follow their choice of algorithms and use Barrett reduction [6] to reduce the results of addition and subtraction, and Montgomery multiplication [20] for multiplication followed by reduction. We also follow the setting in NewHope for the choice of parameters: The modulus parameter q is 12289, and the input size n is 1024. The input is an array of integers whose values fit in 14 bits. Modular-reduction algorithms take one or two 16-bit values and compute a 14-bit output.

2.2 NTT code generation in the tagless-final style

In our previous work [19], we introduced a code-generation framework for NTT, based on the tagless-final style [9]. Since this work builds heavily on our code-generation framework, this section gives a brief introduction to that work.

The tagless-final style is a way to realize a typed DSL via embedding into a typed host language [9]. It uses abstraction facilities in host languages, such as Haskell type classes or the ML module system, to define the syntax of the DSL parameterized by an abstract type for the DSL semantics. Different type class instances or implementations of the module signature give distinct interpretations of a single DSL program[2].

Our code generator is parameterized in two ways: The first one is the semantics of DSL, which follows the standard practice of the tagless-final style. The second one is the semantics of the arithmetic domain the NTT program operates on. We call the first abstraction the **language abstraction** and the second one the **domain abstraction**. Both abstractions are represented in the ML-module system described below.

The language abstraction represents the DSL syntax by a module signature, and its semantics by a module structure. The following module signature `C_lang` represents the syntax of our DSL for generating code in the programming language C. The DSL has sufficient constructs for expressing the NTT algorithm in this paper.

```
module type C_lang = sig
  type 'a expr
  type 'a stmt = 'a expr
  val int_ : int -> int expr    (* constant *)
  val (%+) : int expr -> int expr -> int expr
  ...
  val for_ : int expr -> int expr -> int expr -> (int expr -> unit stmt)
             -> unit stmt
  ...
end
```

The domain abstraction is represented by the following signature:

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

Using these signatures, we describe the innermost loop of Algorithm 1 as follows:

[2] In the module system of ML-family languages, a *signature* is an interface of a module, and a *structure* is its implementation.

```
for_ (int_ 0) m_half (int_ 1) (fun j ->
    let index = k %+ j in
    let omega = arr_get prim_root_powers (coeff_offset %+ j) in
    let2
      (arr_get input index)
      (D.mul (arr_get input (index %+ m_half)) omega)
      (fun u t ->
         seq
           (arr_set input index (D.add u t))
           (arr_set input (index %+ m_half) (D.sub u t))))
```

`arr_get` and `arr_get` are array access and assignment, respectively. `let2 V1 V2 (fun t u -> V3)` is syntactic sugar for the doubly-nested let binding: `let t = V1 in let u = V2 in V3`. The variable `prim_root_powers` stores precomputed twiddle factors in an array. All modular-arithmetic operations are performed by the module `D` which implements the `Domain` signature.

The meaning of this program depends on concrete instantiations of the two abstractions. For this work, we use the term **interpretation** to refer to a concrete instantiation of the language abstraction, and **domain implementation** to refer to a corresponding one for the domain abstraction.

The behavior of DSL programs is determined by giving an interpretation of the module signature `C_lang` including the type `'a expr`. In our previous work, the type `'a expr` is interpreted as an OCaml `string`, since their purpose was solely to generate C programs. For instance, a DSL term `for_` is translated to the string representation of the for loop in the C language. In this work, we use another interpretation that evaluates DSL terms in OCaml by interpreting the type `'a expr` as `'a` as follows:

```
module R = struct
  type 'a expr = 'a
  let int_ n = n
  let (%+) x y = x + y
  ...
  let for_ low high step body =
    let index = ref low in
    for _ = 0 to (high - low) / step - 1 do
      body !index;
      index := !index + step
    done
  ...
end
```

When the NTT program is instantiated with this interpretation, we can directly *execute* the program under the normal semantics for OCaml. The output depends on the domain implementation. The most canonical one, also used in C-code generation, is the domain of integers modulo q with low-level implementations of modular reductions. But we can also use entirely different domains for analysis or verification purposes. For example, we can lift implementations of modular reductions to the domain of *intervals* of integers modulo q: This lets us analyze the NTT program to verify the absence of integer overflow, as we will discuss in Section 4. Similarly, by swapping in a domain representing purely symbolic operations on a finite field, the NTT program would be able to compute a polynomial representation of the outputs with respect to symbolic inputs. Such a highly abstract representation of the NTT computation facilitates verification of functional correctness, as discussed in Section 5.

3 Verification tasks and strategy

Before going into our technical contributions, we summarize the verification tasks at hand and our strategy for tackling them.

To generate a highly efficient C program, our NTT program contains various low-level tricks that make it vulnerable to subtle errors. We highlight several issues that are particularly unique to our program, using the pseudocode of the innermost loop of the NTT program shown in Algorithm 2. The pseudocode differs from Algorithm 1 in that we use low-level modular reductions `barrett_reduce` and `montgomery_multiply_reduce` for Barrett reduction and Montgomery multiplication, respectively. We have also introduced an optimization technique called lazy reduction [3,19] for Barrett reduction.

Algorithm 2 The pseudocode for the innermost loop

```
for ($j = 0$; $j < m/2$; $j = j + 1$) do
    $u = a[k + j]$
    $t$ = montgomery_multiply_reduce($a[k + j + m/2], \Omega[o + j]$)
    if $s \bmod 2 == 0$ then
        $a[k + j]$ = barrett_reduce($u + t$)                    ▷ lazy reduction
    else
        $a[k + j] = u + t$
    end if
    $a[k + j + m/2]$ = barrett_reduce($u + 2q - t$)
end for
```

Highly non-trivial implementation of modular reductions Given a 16-bit integer, `barrett_reduce` computes a value that is congruent to the input and fits in 14 bits[3]. `montgomery_multiply_reduce` multiplies 16-bit and 14-bit integers and reduces the product to fit in 14 bits. To be efficient and safe against timing attacks, these algorithms are implemented in a tricky way. Listing 1 shows a C implementation of Montgomery multiplication. They rely on the instructions `mullo` (and `mulhi`, resp.) to compute the lower 16 bits (and the upper 16 bits, resp.) of the 32-bit product, to keep all intermediate values within 16 bits[4]. The implementation is carefully constructed to make sure that an occasional carry bit is correctly accounted and the output is guaranteed to fit in 14 bits. The latter requirement is satisfied by inserting conditional subtraction `csub`, which subtracts the modulus parameter q from its argument if it is greater than or

[3] We use 12289, which fits in 14 bits, as the modulus parameter q (See Section 2.1).

[4] This is for maximizing parallelism from vectorization.

equal to q, and returns it otherwise. `csub` computes such a value in constant time[5].

```
uint16_t csub(uint16_t arg0) {
  int16_t v_0 = ((int16_t)arg0 - Q);
  return (uint16_t)(v_0 + ((v_0 >> 15) & Q));
}

uint16_t montgomery_multiply_reduce(uint16_t x, uint16_t y) {
  uint16_t mlo = mullo(x, y);
  uint16_t mhi = mulhi(x, y);
  uint16_t mlo_qinv = mullo(mlo, Q_INV);
  uint16_t t = mulhi(mlo_qinv, Q);
  uint16_t has_carry = mlo != 0;
  return csub(mhi + t + has_carry);
}
```

Listing 1: Montgomery multiplication implemented in C (non-vectorized version)

Subtraction in unsigned integers Since data values in our generated code are unsigned integers, we need to be careful with subtraction. to avoid underflow. We need to add to the first operand a multiple of q that is greater than the second operand. We must also ensure that this addition never causes overflow. For our choice of q, the correct multiple of q meeting these conditions turned out to be $2q$.

Lazy reduction As observed in the work on NewHope [3], we do not have to apply Barrett reduction after every addition: Since the result of adding two 14-bit values fits in 15 bits, in the next stage we can add two 15-bit values without the risk of 16-bit overflow. Therefore, Barrett reduction only has to be applied at every other stage. Section 4.3 will show that we can further eliminate Barrett reductions. A more aggressive optimization makes the generated code more vulnerable to integer overflow.

End-to-end verification We are not merely interested in verifying individual pieces of low-level code: The computation of the innermost loop shown in Algorithm 2 is executed $O(n \log n)$ times over an entire execution of NTT, where $n = 1024$ in our case. Our goal is to show that such an accumulated computation gives rise to the value that is equivalent (modulo q) to the one computed by the DFT formula (1).

In this work, we consider both safety and functional correctness. In particular, for the safety aspect, we consider the problem of verifying the absence of integer

[5] In cryptography implementations, being constant-time refers to having no data-dependent control flow, which can become a security hole for timing attacks.

overflow, and for the functional correctness, we consider the equivalence of the NTT program against DFT. We consider the safety aspect separately because (1) it simplifies the latter task and (2) interval analysis we develop for verifying the absence of integer overflow uncovers a new optimization opportunity. So we believe our safety verification is of independent interest.

For verifying functional correctness, we do not pursue an approach using an interactive proof assistant such as Coq, which can give us the highest level of correctness guarantee. Since we aim at generating and verifying highly efficient cryptographic code whose implementation strategy changes frequently, we stick to a lightweight approach that allows one to change the implementation and adapt the verification component quickly and easily.

Thus, we have developed a dedicated procedure for our verification problem. Our approach works on a DSL program, not on the generated C program. But the DSL program contains all low-level details that are present in the C program, so our verification procedure takes all of such details into account. Thus, correctness assurance we give to the DSL program directly translates to the generated C program[6]. We have found that an attempt to prove functional correctness of the whole program all at once is intractable: Instead, our overall strategy for end-to-end verification is based on decoupling low-level components in the NTT program from the high-level aspect of the NTT algorithm. Verification of low-level components can be done straightforwardly, while we developed a simple and effective verification procedure to show the equivalence of the NTT program and the DFT formula in a purely mathematical setting. The decision to do verification at the DSL level and the highly parameterized nature of our DSL program make such decoupling and subsequent verification possible.

4 Interval analysis on the NTT program

To verify the absence of integer overflow, we present a simple interval analysis as part of a program-generation framework for NTT programs. We have implemented our own analyzer, rather than using an off-the-shelf tool for C programs, to exploit domain-specific knowledge and compute more precise bounds than the ones computed by the latter tools such as Frame-C [7]. We will show that our analysis not only verifies the absence of integer overflow but also allows us to derive a new optimization that was not known previously.

4.1 Modular arithmetic on intervals

We have designed an abstract interpreter for our modular-arithmetic routines, building on the two abstractions we described in Section 2.2: We use the interpretation of DSL that evaluates DSL terms directly in OCaml, and the set

[6] For simplicity, we do not consider the effect of vectorization for our verification purpose, although the generated program is fully vectorized with multiple SIMD instruction sets. All of the low-level issues that motivate our verification effort are manifested in the non-vectorized implementation.

of intervals (`low`, `high`) as our domain implementation where `low` and `high` are integers representing the lower and upper bounds, respectively. The `Domain` module in Section 2.2 is instantiated to the following structure:

```
module IntegerModulo_interval : Domain = struct
  type t = int * int
  let add (x1, y1) (x2, y2) = ...
  let sub (x1, y1) (x2, y2) = barrett_reduce([x1 + 2Q - y2, y1 + 2Q - x2])
  let mul (x1, y1) (x2, y2) = ...
end
```

Simple operations such as addition can be directly lifted to the intervals, building on the standard definition of interval arithmetic. Montgomery multiplication, represented by `mul` above, is lifted to intervals by composing basic operations, such as `mullo` and `mulhi`, lifted to the interval domain.

Lifting Barrett reduction to interval domains requires more care. As shown below, Barrett reduction requires only three operations.

```
uint16_t barrett_reduce(uint16_t x) {
  uint16_t v = mulhi(x, 5);
  return x - mullo(v, Q);
}
```

We could have lifted Barrett reduction by composing the interval version of high product, low product, and subtraction. But this approach faces difficulty in the subtraction `x - mullo(v, Q)`: Its second operand is the result of low product, which, when lifted to intervals, always results in the least precise range [0, 65535]. Even though the first operand x is always greater than the second one[7], it cannot be automatically inferred by applying interval analysis naively. To get maximally precise bounds, we lift Barrett reduction to intervals by applying the integer domain operation to all integers in the input interval, and taking the minimum and maximum of the results of these operations. This comes at the high cost of runtime, but since the input to Barrett reduction is at most 16 bits, it does not significantly slow down the analysis[8].

4.2 Verifying bounds

Each low-level modular-arithmetic operation has certain conditions on its inputs and output that need to be satisfied. We formulate these conditions as assertions to be checked during interval analysis, summarized in Table 1.

For example, the second operand of addition and subtraction has a tighter bound of `max_uint14`, because it is the result of modular multiplication which must fit in 14 bits where `max_uint14` refers to the maximum of unsigned 14-bit integers, namely $(1 \ll 14) - 1$. Similarly for `max_uint15` and `max_uint16`.

[7] `mullo(mulhi(x,5),q)` is less than $\left\lfloor x\frac{1}{q}\right\rfloor q$, since $5q < 65535$ for our choice of q.

[8] It took only a few seconds for the input of size 1024.

The bound of `max_uint15` on the first operand is due to lazy reduction. Bounds in Table 1 in turn depend on the validity of bounds on Barrett reduction and conditional subtraction, shown in Table 2.

Table 1. Pre/Post-conditions for input $[x_1, y_1], [x_2, y_2]$ and output $[x_3, y_3]$

Operations	Precondition	Postcondition
`add`	$y_1 \leq$ `max_uint15` $\wedge\ y_2 \leq$ `max_uint14`	$y_3 \leq$ `max_uint16`
`sub`	$y_1 \leq$ `max_uint15` $\wedge\ y_2 \leq$ `max_uint14`	$y_3 \leq$ `max_uint14`
`mul`	$y_1 \leq$ `max_uint15` $\wedge\ y_2 < q$	$y_3 \leq$ `max_uint14`

Table 2. Pre/Post-conditions for input $[x_1, y_1]$ and output $[x_2, y_2]$

Operations	Precondition	Postcondition
`barrett_reduce`	$y_1 \leq$ `max_uint16`	$y_2 \leq$ `max_uint14`
`csub`	$y_1 < 2q$	$y_2 < q$

For Barrett reduction, we need to verify an additional assertion saying that the first argument of the final subtraction is not smaller than the second argument. This is realized by inserting an assertion as follows:

```
let barrett_reduce x =
  ...
  let rhs = ...
  assert (x >= rhs);
  let res = x - rhs in
  ...
```

Note that the assertion is inserted in the structure `IntegerModulo_interval` only. We do not have to modify the DSL program, because it is parameterized with respect to domain interpretations.

We have confirmed that, given an array of intervals $[0, q-1]$ as input, all of our assertions are not violated. Hence, there is no possibility for integer overflow for our code.

We have also conducted the same verification experiment on the generated C code using the Frama-C value analysis plugin [7]. For this purpose, we added the above assertions as ACSL specifications [1] to the generated C code. Frama-C was able to verify all but two assertions: the postcondition in Table 2 and the assertion on the bound before the final subtraction[9]. We suspect that this outcome arises from directly translating Barrett reduction on integers to intervals by composing interval operations, which, as we observed in Section 4.1, can lead to a loss in precision.

4.3 Improving lazy reduction

During the course of interval analysis in the previous subsection, we found a way to optimize the generated code even further: Barrett reduction after addition,

[9] We have chosen options that maximize the precision of the analysis.

which we refer to as lazy Barrett reduction for brevity, needs to be applied only once in **three** stages, rather than every other stage as we adopted from NewHope. Realization of lazy reductions comes from the following observations:

- An operation that is most vulnerable to unsigned overflow is the addition of $2q$ in subtraction, $(x + 2q) - y$. Since x is an unsigned 16-bit integer, the maximum value that x can take without causing overflow in the addition is $65535 - 2q = 65535 - 2 * 12289 = 40957$, where 65535 is the maximum value of an unsigned 16-bit integer.
- Our analysis showed that the maximum value that an input to lazy Barrett reduction can take is 39319.

The first observation suggests that there is no need to apply Barrett reduction before the value reaches 40957, while from the second one we know that the input to lazy Barrett reduction is at most 39319. Therefore, we can omit one more reduction before we need to reduce the value to 14 bits. Since each stage has 512 additions, and we have reduced the number of stages where lazy Barrett reduction is applied from 5 to 3, in total we are able to remove $2 * 512$ Barrett reductions. The actual speedup over our previous work is summarized in Table 3. On AVX2, the improved lazy reduction brought good speedup (14%) compared to the baseline, while on AVX512 the speedup is modest (1.5%).

Table 3. Speedup by the improved lazy reduction (CPU: Intel Core i7-1065G7)

	Cycle counts	Speedup
AVX2 baseline	5398	
AVX2 backend + improved lazy reduction	4744	14%
AVX512 baseline	4381	
AVX512 backend + improved lazy reduction	4317	1.5%

The interval estimated by Frama-C is not precise enough to derive the same conclusion as above: Frama-C computed the maximum value an input to lazy Barrett reduction can take to be 40959, which is slightly bigger than the hard threshold of 40957 required for safely enabling the optimization above. This difference in bounds comes from the increased precision in our implementation of lifted Barrett reduction: Our analysis shows that the maximum value after interval subtraction is 14743, in contrast to 16383 computed by Frama-C. The difference in the precision, $16383 - 14743$, is equal to $40959 - 39319$, that is the difference in the maximum values an input to lazy Barrett reduction can take.

As a sanity check, we confirmed that our analysis fails to verify the assertions if we omit one more Barrett reduction from our code. We also tested the generated C program with the improved lazy reduction on 10000 randomly chosen concrete values as an input, and confirmed that all outputs were correct with respect to the DFT formula, and that each output belongs to the corresponding interval computed by our interval analysis.

5 Verifying functional correctness

The goal is to show that the output computed by the optimized NTT program is equivalent to the one computed by DFT. We first discuss the first attempt which did not work out, and then explain the final solution we developed.

5.1 Naive approach

One straightforward but naive approach is to translate the entire NTT program into a formula in the bit-vector theory [16], and verify using an SMT solver the equivalence of the formula and the one obtained from the DFT formula. The translation to a formula is easily done by symbolically computing the NTT program in the bit-vector theory.

This approach did not work, since the resulting formulas were so large that Z3, the SMT solver we used, did not terminate after more than six hours and before it ran out of memory. We also tried replacing complicated implementations of modular reductions with the naive ones using the modulo operator (`bv_urem` in SMT-LIB), but the end-to-end verification was still not tractable.

5.2 Decomposition of verification task

A natural idea to overcome the difficulty of verifying a program like optimized NTT, which has both low-level details and a high-level algorithmic structure, is to decompose the original verification problem into several components, in a way that separate verification of each component would imply functional correctness of the whole program. We give an overview of the decomposition here; a more detailed account on the whole verification process is shown in Appendix A.

Recall the pseudocode in Algorithm 2. Our interval analysis in Section 4 has shown that, on an end-to-end execution of the NTT program, there will be no possibility of integer overflow. This means that, to verify program equivalence modulo q, we can replace lazy reduction by an eager one that always applies Barrett reduction after addition. This simplifies the original psuedocode on the left of Fig. 1 to the one on the right.

```
$u = a[k + j]$
$t =$ montgomery_multiply_reduce$(\dots)$
if $s \bmod 2 == 0$ then
    $a[k + j] =$ barrett_reduce$(u + t)$
else
    $a[k + j] = u + t$
end if
$a[k + j + m/2] =$ barrett_reduce$(u + 2q - t)$
```

```
$u = a[k + j]$
$t =$ montgomery_multiply_reduce$(\dots)$
$a[k + j] =$ barrett_reduce$(u + t)$
$a[k + j + m/2] =$ barrett_reduce$(u + 2q - t)$
```

Fig. 1. Simplifying the lazy reduction (left) to the eager one (right)

The next step for simplification is to replace low-level implementations of modular arithmetic with much simpler operations. For this purpose, we need to prove correctness of Barrett reduction and Montgomery multiplication by `(u + t) % q` = `csub(barrett_reduce(u + t))` for Barrett reduction and similarly for Montgomery multiplication[10]. We describe our verification procedure in Section 5.3. The simplified arithmetic operations, represented by $+'$, $-'$, and $*'$ in Fig. 2, are interpreted as symbolic operations on a finite field with built-in modular arithmetic.

```
$u = a[k + j]$
$t = a[k + j + m/2] *' \Omega[o + j]$
$a[k + j] = u +' t$
$a[k + j + m/2] = u -' t$
```

Fig. 2. Simplified butterfly computation on a finite field

Section 5.4 describes how such symbolic operations facilitate equivalence checking against the DFT formula. Since all low-level concerns have been resolved until this point, we can focus on the algorithmic aspect of the NTT program.

5.3 Verifying modular-reduction algorithms

We have verified the equivalence of Barrett reduction and Montgomery multiplication implementations against the naive approach of using a built-in modulo operator (the `%` operator in C). We encode both approaches into Z3 formulas using the bit-vector theory, and check their equivalence. For example, Montgomery multiplication is implemented in the DSL as follows:

```
let montgomery_multiply_reduce x y =
  let mlo = mullo x y in
  let mhi = mulhi x y in
  let mlo_qinv = mullo mlo (const Param.qinv) in
  let t = mulhi mlo_qinv (const Param.q) in
  let carry = not_zero mlo in
  let res = mhi %+ t %+ carry in
  csub res
```

Listing 2: Montgomery multiplication implementation from [19]

We provide an implementation of the domain abstraction that, together with the direct evaluation of DSL terms by the host language, translates the DSL expression into a bit-vector formula. All DSL constructs required for Montgomery multiplication have a direct counterpart in the bit-vector theory, except for the high-product instruction `mulhi` which can be emulated easily[11].

[10] The symbol = represents the exact equality on integers. The additional conditional subtraction is necessary since the outputs of Barrett reduction can be larger than q.
[11] Refer to our source code for details on the translation from DSL to Z3 formulas.

By these ingredients, we can apply Z3 to verify Montgomery multiplication. More concretely, let `opt_formula` and `ref_formula` be the Z3 formulas for the implementation in Listing 2 and the naive multiplication followed by a modulo operation, respectively. We ran Z3 to check unsatisfiability of the formula `opt_formula` $\neq$ `ref_formula`, which has been successful. Similarly, correctness of the Barrett reduction has been proved using Z3.

5.4 Proving correctness of the simplified NTT program

Our strategy for verifying the simplified NTT program is based on the following observation: Since DFT is a linear transformation, each output element can be represented as a linear polynomial on input variables. Since NTT also represents a linear transformation, we only have to prove that all coefficients on each variable in the two polynomials coincide up to congruence[12]. Therefore, we symbolically execute the simplified NTT program to compute such a linear polynomial for all output elements, and test if all coefficients are congruent to the corresponding rows of the DFT matrix. Thanks to the simplification stated before, the polynomial is truly linear in the sense that it consists of addition, subtraction, and multiplication by a constant (no explicit modulo operations) only. We can extract coefficients from the polynomials and compare them.

To make this idea concrete, we introduce a domain implementation that represents symbolic computations on polynomials:

```
module D_symbolic : Domain = struct
  type exp =
    | Const of int
    | Sym of string
    | Add of exp * exp
    | Sub of exp * exp
    | Mul of exp * exp

  type t = exp

  let add x y = Add(x, y)
  ...
end
```

When we symbolically execute the NTT program using this domain, with an array of symbolic integers (represented by `Sym` constructor of `exp` type above) as an input, we end up with an output array whose i-th element represents all computations that contribute to the i-th output. For each output, we simplify such a nested polynomial expression to obtain a linear polynomial, and compare the coefficients array with the corresponding row of the DFT matrix. We have confirmed that, for all output elements, our verification succeeded in establishing congruence of the NTT outputs and the DFT matrix.

[12] The coefficients computed by the NTT program may contain negative values due to subtraction in the butterfly operation.

Note how the two abstractions, the language and domain abstractions, simplified equivalence checking via symbolic computation: By composing the language interpretation that evaluates DSL terms directly in the host language, and the domain implementation representing symbolic operations, symbolic computation of NTT is naturally realized. Such a high degree of program abstraction would be nearly impossible if we would have operated on the low-level C program.

6 Discussion

Why a "unified" approach? In a traditional approach where generated code is verified directly, one has to reconstruct the original high-level structure in a DSL program from the generated low-level program, before doing any kind of analysis or verification. Even if such reconstruction was possible, we believe that the kind of program abstraction we rely on, such as the reinterpretation of the DSL program for symbolically computing polynomials, is extremely hard to accomplish automatically. In our unified approach, a verification procedure starts with a high-level DSL program. This makes the verification task simpler and paves a way for verifying more challenging properties than those handled by off-the-shelf automatic tools. At the same time, since program generation is based on the same DSL program, all interesting low-level concerns in the generated program are taken into account during verification.

The downsides of our approach are in (1) not verifying the generated code directly and (2) relying on unconventional trusted base[13]. Since we regard verification of the DSL program as a proxy for verification of the generated program, there is always a question on the gap between what is generated and what is verified. In addition, our approach assumes that our implementation of the two DSL interpretations, the program generator and the verifier, correctly respects the semantics of the original DSL program[14].

Despite the major disadvantages above, we believe that our approach is a promising step toward verifying functional correctness of a low-level, highly optimized program. We view the pros and cons of our approach as a trade-off between more possibilities for verification and larger trusted base.

7 Related Work

Earlier work on verifying FFT focused on establishing equivalence of a textbook-style, recursive formulation of the FFT algorithm against DFT using a proof assistant [12,2,8]. Recently, Navas et al. [21] verified the absence of integer overflow in an implementation of NTT[17]. However, verification of functional correctness

[13] See Appendix A for our trusted base.

[14] However, note that both interpretations are based on the tagless-final style and thus they operate on DSL constructs at the most primitive level (such as translating the DSL for loop to that of OCaml or C). Therefore, we believe that their correctness is a reasonable assumption.

has been largely left open. To the best of our knowledge, there has been no prior work on verifying functional correctness of a highly optimized NTT implementation.

Verified implementations of low-level cryptographic primitives have been an active research topic [4,11,26,22]. Existing work targets those primitives that are already used widely, for example those in the Internet protocols, and proposes optimized implementations that are thoroughly verified using Coq or F*. We think that such full-scale verification is realistic only for primitives that are important today and whose implementations are less stable, due to its high cost: For bleeding-edge primitives such as NTT, more lightweight approaches like ours would be more accessible and useful for practitioners.

Outside of cryptography, the pioneering work by Amin et al. [5] considered correctness issues in the context of staging and generative programming. Their approach is based on generating C code together with correctness contracts as ACSL specifications, which can be verified by an external tool. Since they verify the generated code directly, they do not have to trust the code generator or a verifier for the DSL program. Although the approach of Amin et al. has a major advantage in this respect, what they can verify are fundamentally limited by the capability of external tools operating on the generated C program. For example, the case studies in [5] are limited to verifying safety properties such as memory safety of an HTTP parser or functional correctness of simple programs such as sorting. Our approach is complementary to theirs in the sense that we can verify more challenging properties in exchange for bigger trusted computing base.

8 Conclusion

We have proposed an approach for giving correctness assurance to the generated code in the context of generative programming. Integration of code generation and verification under one DSL framework enabled us to (1) incorporate an abstract interpretation to prove, for instance, the absence of integer overflow, and (2) decompose the end-to-end correctness verification problem into low-level and high-level parts, each of which can be verified separately. Our approach is lightweight in the sense that we make use of automation via abstract interpretation and symbolic computation. We have applied our approach to a highly optimized implementation of NTT, which is a key building block of next-generation cryptographic protocols, and successfully verified its functional correctness.

For future work, we plan to generalize our approach so that existing schemes other than NewHope or new ones can be similarly reimplemented and verified. We are also interested in exploring whether our unified approach is applicable to other domains, outside of NTT or cryptography.

Acknowledgments. We thank Hiroshi Unno for the helpful discussion. Feedback from anonymous reviewers helped improve this paper and is greatly appreciated. The second author is supported in part by JSPS Grant-in-Aid for Scientific Research (B) 18H03218.

References

1. ANSI/ISO C specification language. https://frama-c.com/html/acsl.html
2. Akbarpour, B., Tahar, S.: A methodology for the formal verification of FFT algorithms in HOL. In: Hu, A.J., Martin, A.K. (eds.) Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3312, pp. 37–51. Springer (2004). https://doi.org/10.1007/978-3-540-30494-4_4
3. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key exchange: A new hope. In: Proceedings of the 25th USENIX Conference on Security Symposium. p. 327–343. SEC'16, USENIX Association, USA (2016)
4. Almeida, J.B., Barbosa, M., Barthe, G., Blot, A., Grégoire, B., Laporte, V., Oliveira, T., Pacheco, H., Schmidt, B., Strub, P.Y.: Jasmin: High-assurance and high-speed cryptography. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. p. 1807–1823. CCS '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3134078
5. Amin, N., Rompf, T.: LMS-Verify: abstraction without regret for verified systems programming. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 859–873. ACM (2017). https://doi.org/10.1145/3009837.3009867
6. Barrett, P.: Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In: Proceedings on Advances in Cryptology—CRYPTO '86. p. 311–323. Springer-Verlag, Berlin, Heidelberg (1987)
7. Bühler, D.: Structuring an Abstract Interpreter through Value and State Abstractions:EVA, an Evolved Value Analysis for Frama-C. (Structurer un interpréteur abstrait au moyen d'abstractions de valeurs et d'états :Eva, une analyse de valeur évoluée pour Frama-C). Ph.D. thesis, University of Rennes 1, France (2017), https://tel.archives-ouvertes.fr/tel-01664726
8. Capretta, V.: Certifying the Fast Fourier Transform with Coq. In: Boulton, R.J., Jackson, P.B. (eds.) Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2152, pp. 154–168. Springer (2001). https://doi.org/10.1007/3-540-44755-5_12
9. Carette, J., Kiselyov, O., Shan, C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. **19**(5), 509–543 (2009). https://doi.org/10.1017/S0956796809007205
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition. The MIT Press, 3rd edn. (2009)
11. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019. pp. 1202–1219. IEEE (2019). https://doi.org/10.1109/SP.2019.00005
12. Gamboa, R.A.: The Correctness of the Fast Fourier Transform: A Structured Proof in ACL2. Form. Methods Syst. Des. **20**(1), 91–106 (Jan 2002), https://doi.org/10.1023/A:1012912614285
13. Güneysu, T., Oder, T., Pöppelmann, T., Schwabe, P.: Software Speed Records for Lattice-Based Signatures. In: Gaborit, P. (ed.) Post-Quantum Cryptography. pp. 67–82. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

14. Kiselyov, O., Biboudis, A., Palladinos, N., Smaragdakis, Y.: Stream fusion, to completeness. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. p. 285–299. POPL 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009880
15. Krishnaswami, N.R., Yallop, J.: A typed, algebraic approach to parsing. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 379–393. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314625
16. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View, Second Edition. Texts in Theoretical Computer Science. An EATCS Series, Springer (2016). https://doi.org/10.1007/978-3-662-50497-0
17. Longa, P., Naehrig, M.: Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In: Foresti, S., Persiano, G. (eds.) Cryptology and Network Security. pp. 124–139. Springer International Publishing, Cham (2016)
18. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Gilbert, H. (ed.) Advances in Cryptology – EUROCRYPT 2010. pp. 1–23. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
19. Masuda, M., Kameyama, Y.: FFT program generation for Ring LWE-based cryptography. In: Nakanishi, T., Nojima, R. (eds.) Advances in Information and Computer Security. pp. 151–171. Springer International Publishing, Cham (2021)
20. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation **44**, 519–521 (1985)
21. Navas, J.A., Dutertre, B., Mason, I.A.: Verification of an optimized NTT algorithm. In: Christakis, M., Polikarpova, N., Duggirala, P.S., Schrammel, P. (eds.) Software Verification. pp. 144–160. Springer International Publishing, Cham (2020)
22. Protzenko, J., Parno, B., Fromherz, A., Hawblitzel, C., Polubelova, M., Bhargavan, K., Beurdouche, B., Choi, J., Delignat-Lavaud, A., Fournet, C., Kulatova, N., Ramananandro, T., Rastogi, A., Swamy, N., Wintersteiger, C.M., Zanella-Beguelin, S.: Evercrypt: A fast, verified, cross-platform cryptographic provider. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 983–1002 (2020). https://doi.org/10.1109/SP40000.2020.00114
23. Seiler, G.: Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. IACR Cryptol. ePrint Arch. **2018**, 39 (2018)
24. Shaikhha, A., Klonatos, Y., Koch, C.: Building efficient query engines in a high-level language. ACM Trans. Database Syst. **43**(1) (Apr 2018). https://doi.org/10.1145/3183653
25. Wei, G., Chen, Y., Rompf, T.: Staged abstract interpreters: fast and modular whole-program analysis via meta-programming. Proc. ACM Program. Lang. **3**(OOPSLA), 126:1–126:32 (2019). https://doi.org/10.1145/3360552
26. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: HACL*: A verified modern cryptographic library. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. p. 1789–1806. CCS '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3134043

Appendix A Programs to be verified and their semantics

The verification procedure in Section 5 is a series of step-by-step simplifications of programs and their correctness proofs. The following table lists the programs and the domain interpretations in the procedure.

	program	domain	arithmetic operation
P_0	DFT formula (1)	Z_q	arithmetic operations in Z_q
P_1	DSL program in Section 2.2	Z_q	arithmetic operations in Z_q
P_2	the same as P_1	unsigned int	arithmetic with modulo-q
P_3	the same as P_1	unsigned int	low-level operations
P_4	P_1 + lazy reduction	unsigned int	low-level operations
P_5	generated C code	unsigned int in C	arithmetic operations in C

P_0 is the DFT formula (1) in Section 2.1. P_1, P_2, and P_3 are the DSL program whose inner-most loop was given in Section 2.2 with different domain interpretations. For the interpretation of DSL, we take the natural 'interpreter' semantics, which is essentially the same as the module `R` in Section 2.2.

P_1, P_2, and P_3 differ in the domain interpretations. For P_1, the domain is interpreted as Z_q. For P_2, the domain is interpreted as the set of 16bit unsigned integers, and the arithmetic operations are those for unsigned integers followed by the modulo-q operation. To treat multiplication within 16 bits, we use `mullo` and `mulhi` in Section 3. For P_3, the domain remains the same as P_2, while the arithmetic operations are replaced by low-level operations such as Barrett reduction. The semantics of unsigned integers and their operations is specified by the bit-vector theory [16]. P_4 is the same as P_3 except that it employs lazy reduction in Section 3.

P_5 is the C code generated by interpreting the DSL constructs as generators for strings that represent the corresponding C code. This process (called *off-shoring* in the literature) is conceptually a trivial injection, however, formalizing it involves the semantics of the C language and is beyond the scope of this paper, and we put the equivalence of P_4 and P_5 into our trusted base.

Besides it, our trusted base includes correctness of our interval analysis, symbolic computation, and the implementations of helper functions such as `mullo` and `mulhi`. With this trusted base as well as the language and domain interpretations explained above, this paper has verified that, for $0 \leq k \leq 3$, P_k is extensionally equal (modulo q) to P_{k+1} (written $P_k =_{ext} P_{k+1}$): $P_3 =_{ext} P_4$ and $P_1 =_{ext} P_2$ in Section 4, $P_2 =_{ext} P_3$ in Section 5.3, and $P_0 =_{ext} P_1$ in Section 5.4.