IACR Communications in Cryptology
Vol. 0, No. 0, 25 pages.

Formally Verified Number-Theoretic Transform

Alix Trieu

ANSSI, Paris, France

Abstract. In recent years, the number-theoretic transform (NTT) has become increasingly common in cryptography, in part due to multiple lattice-based cryptographic schemes being selected for standardization during the NIST PQC competition. Indeed, polynomial multiplications are one of the most computing intensive operations in these schemes and the NTT is crucial in decreasing the performance cost. The NTT also appears in other areas such as fully homomorphic encryption (FHE) and zero-knowledge proofs (ZKP) which are increasingly used in privacy-preserving applications. In this paper, we show how to formally specify the NTT in the Rocq proof assistant, and how we used this specification to automatically derive formally verified implementations of both complete and incomplete NTTs for multiple cryptographic schemes.

Keywords: High-assurance cryptography · NTT · Fiat-Crypto · Rocq

1 Introduction

With potentially "cryptographically relevant" quantum computers on the horizon, multiple authorities have published roadmaps to face the post-quantum cryptography (PQC) transition. As part of this process, the U.S. National Institute of Standards and Technology (NIST) has organized a competition to select and standardize new cryptographic schemes that can stand up to the quantum threat. This has resulted in NIST selecting one key-encapsulation mechanism (KEM) and three digital signature algorithms (DSA) in 2022. Three of those schemes were standardized in 2024 as ML-KEM [Nat24b], ML-DSA [Nat24a] and SLH-DSA [Nat24c], while the last one will be standardized as FN-DSA.

These schemes follow different paradigms from pre-quantum cryptographic schemes (e.g., elliptic curves). As such, they also use different mathematical operations, and there is less experience in implementing them efficiently and correctly.

Despite this, due to the omnipresence of cryptography in all of society, and the amount of data it is applied to, performance is of paramount importance. It was therefore one of the main criteria used for selecting schemes during the competition. It may thus not be surprising that all the lattice-based cryptographic schemes among the NIST PQC competition winners use a so-called Number-Theoretic Transform (NTT) in order to speed up modular polynomial multiplications. Interestingly, NTT is not solely regarded as a possible optimization to implement, but is regarded as mandatory to implement as specified in the published standards [Nat24b, Nat24a].

Lattices also form the basis for fully homomorphic encryption (FHE) [Gen09], which is used to ensure digital privacy by allowing computing over encrypted data for applications such as cloud computing. Moreover, verifiable distributed aggregation functions (VDAF) [BCPS25] such as Prio3 [CB17] use a form of zero-knowledge proofs (ZKP), which necessitates fast polynomial interpolation that can be provided by the NTT.

This shows that the NTT has become an ubiquitous operation in cryptography. However, it is a quite costly operation and has thus been a target for much optimization. Due to the

E-mail: alix.trieu@ssi.gouv.fr (Alix Trieu)

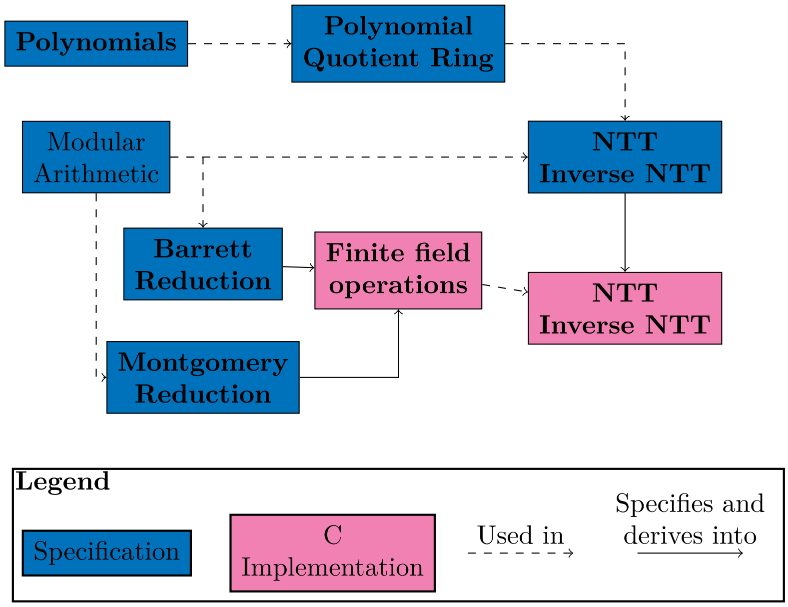

Figure 1: Description of the framework. Contributions in bold.

complex nature of NTT, making sure that the optimizations are correctly implemented is a difficult endeavour. Because of the inherent nature of cryptography, exhaustively testing all inputs is infeasible.

Formal verification is a solution to this issue, and can be used to formally prove that an implementation correctly behaves as mathematically specified for all inputs. *Computer-aided cryptography* [BBB+21] is the application of such *formal methods* to cryptographic engineering. This has resulted in the past decade in many high-speed and high-assurance cryptographic implementations being created, which are now being used in production. For instance, major web browsers such as Google Chrome and Firefox make use of formally verified cryptographic software that are included in the BoringSSL and NSS libraries [EPG+19, ZBPB17].

However, embedded systems are also becoming increasingly important as they are used in areas such as IoT, automotive, medical, etc. There is therefore a major need to secure them. As micro-controllers are often low power, they do not always support the features needed to be able to run the same cryptographic implementations as those used in a browser running on a general purpose computer. We thus aim to produce formally verified *portable* code that does not feature specialized instructions.

While it may seem attractive to use different specialized tools to tackle specific verification challenges, subtle differences in the specifications used for each tool may appear and thus threaten the correctness of the whole verification, while largely increasing the Trusted Computing Base (TCB) footprint. We follow a *foundational* approach, meaning that all verification are carried out using a single tool. Our TCB is therefore close to minimal and includes only the tool we use — namely, the Rocq proof assistant — as well as our handwritten specifications of the NTT and the extraction mechanism to C code.

Contributions The deployment of post-quantum cryptography is an opportunity to roll out formally verified implementations from the start. The main contribution of this paper is to provide a formalization of a core operation for lattice-based cryptography, namely the Number-Theoretic Transform (Figure 1), which we show to be general enough to be able to derive formally verified implementations from it.

1. We formally specify a ring theory of polynomials and their quotient rings in the Fiat-Crypto framework using the Rocq proof assistant;
2. We then formally specify the forward and inverse Number-Theoretic Transform and show that these indeed form a ring isomorphism between $\mathbb{Z}_q[X]/(X^n+1)$ and some $\mathbb{Z}_q[X]/(X^m-\zeta^a)\times\cdots\times\mathbb{Z}_q[X]/(X^m-\zeta^b)$ where n is a power-of-two;
3. We formalize and verify the correctness of signed Barrett and Montgomery reduction algorithms;
4. We extract verified constant-time C implementations of the NTT for multiple compatible schemes such as ML-DSA, ML-KEM, and others, thus showing that our specification is general enough to handle different uses;
5. Finally, we evaluate these implementations against hand-optimized implementation.

Outline of this paper Section 2 first discusses how the NTT is specified in standards or specification documents, and how other verification projects have formalized it. It then explains our own formalization motivated by these observations. Next, Sections 3 explains how we refine implementations from the specification and present some optimization techniques for the NTT that we formalized. Section 4 and 5 then explain and show how we used Fiat-Crypto to derive formally verified C implementations of the NTT which we then compare to existing implementations. Finally, Section 6 reviews related works and Section 7 presents possible avenues for future work and concludes.

We believe that Sections 2 and 3 may be of independent interest for those who wish to better understand how the NTT and its implementation work, even if they don't care about formal verification.

Supplementary material All results presented in this paper are formally verified using the Rocq proof assistant and available as supplementary material.

2 Number-Theoretic Transform

One of the most computationally intensive operations in lattice-based cryptography is polynomial multiplication in a quotient ring $\mathbb{Z}_q[X]/(f(X))$ which requires a quadratic number of operations when implemented following the "school-book" approach before reducing modulo $f(X)$.

Parameters are thus often chosen so that a NTT can be applied, which provides polynomial multiplications with an asymptotic loglinear complexity rather than the usual quadratic complexity.

In this section, we explain how we formalized the NTT using the Rocq proof assistant, though we deliberately stay at a mathematical level instead of writing directly Rocq code so that it can be understood by people not necessarily interested in formal verification. It is however explained at a low-level enough so that it should be easily reproducible in other proof assistants such as Easycrypt [BGHZ11], HOL Light [Har09], Lean [dMU21] and others. The written code and proofs are available in the accompanying supplementary material for those interested.

2.1 NTT in Specification Documents and Other Formalizations

Before presenting our formalization of the NTT, we discuss how the operation is specified in standards or specification documents so that we can try to capture as many use-cases as possible with our formalization. We then compare to other formalizations in related works.

ML-DSA [Nat24a, Eq. (2.1)] defines the operation as

$$\mathsf{NTT}(p) = (p(\zeta^{2\mathsf{BitRev}_8(0)+1}), \ldots, p(\zeta^{2\mathsf{BitRev}_8(255)+1}))$$

where ζ is a 512^{th} root of unity and states it is an isomorphism between $\mathbb{Z}_q[X]/\left(X^{256}+1\right)$ and $\prod_0^{255} \mathbb{Z}_q$.

Kyber [SAB+20, Eq. (3)] and ML-KEM [Nat24b, Eq. (4.12)] both specify the NTT as

$$\mathsf{NTT}(p) = (p \bmod X^2 - \zeta^{2\mathsf{BitRev}_7(0)+1}, \ldots, p \bmod X^2 - \zeta^{2\mathsf{BitRev}_7(127)+1})$$

As opposed to the earlier evaluation-based specification, this one is based on the Chinese Remainder Theorem (CRT) [Ber01], though not explicitly written. Moreover, the NTT is "incomplete" contrarily to ML-DSA's "complete" one, and cannot thus be specified similarly. Furthermore, [HYS+25] suggests that there might be beneficial tradeoffs in using incomplete NTT.

While not strictly specified, Prio3 [BCPS25, CB17] makes use of an (inverse) NTT to interpolate a polynomial given values at powers of a root of unity similar to the specification in ML-DSA. However, this can be easily recovered using CRT techniques, since

$$p(a) = p \bmod X - a$$

As such, our formalization of the NTT is based on the CRT.

The NTT can be seen as a special form of the Fast Fourier Transform over finite fields, which has been formalized in earlier works using Rocq [Cap01, Thé22]. However, these works follow an evaluation-based approach by computing $\langle p(\zeta^0), p(\zeta^1), \ldots \rangle$ which is too restrictive as it cannot model incomplete NTT as in ML-KEM. Furthermore, the NTT domains usually follow a bit-reversed order which is incompatible with their formalizations.

There have been few other works on formalizing the NTT in a proof assistant. Kreuzer [Kre24] verifies the correctness and security of Kyber in Isabelle as a cryptography primitive, the author thus also specifies its NTT. However, the algorithm that is formalized is only specific to Kyber and there is no implementation derived from it.

[ABB+23] presents a verified implementation of Kyber in Jasmin [ABB+17], which they recently updated to ML-KEM [AOB+24]. Their works are much more complete than ours as they provide a fully verified implementation of ML-KEM while we only verify the NTT. However, their TCB is much larger than ours as their formalization relies on an unverified translation from Jasmin to EasyCrypt. Furthermore, their formalization seems rather specific to Kyber, as their specification of the NTT uses the explicit formula to compute each coefficient of the resulting NTT array provided by the Kyber specification [SAB+20, Eq. (4-5)] (though they were removed in the ML-KEM specification [Nat24b]).

$$\hat{f}_{2i} = \sum_{j=0}^{127} f_{2j}\zeta^{(2\mathsf{br}_7(i)+1)j} \qquad \hat{f}_{2i+1} = \sum_{j=0}^{127} f_{2j+1}\zeta^{(2\mathsf{br}_7(i)+1)j}$$

As such, they will have to modify the specification and the corresponding proofs for different parameters, e.g., complete NTT as in ML-DSA [Nat24a] or even more incomplete NTT as proposed by [HYS+25]. Furthermore, they cannot take advantage of the algebraic structure of the NTT and have to reprove consequently some (not necessarily difficult) lemmas that are direct consequences of the NTT simply computing polynomial modular reductions, e.g., it being a ring homomorphism. Our formalization avoids these issues as we present next.

2.2 Formalization of the NTT

As explained earlier, the NTT can be seen as a way to speed up polynomial multiplications in a polynomial quotient ring $\mathbb{Z}_q[X]/(f(X))$. As such, we assume that such a theory has already been formalized.

In the case of the NTT, the polynomial modulus is usually chosen to be of the form $f(X) = X^{2^n} + 1$ and prime q is chosen so that a power-of-two primitive root of unity ζ exists in the field $\mathbb{Z}_q$.

Under those conditions, there exists m such that $\zeta^{2^m} = -1$ and the following equations hold:

$$\begin{aligned} X^{2^n} + 1 &= X^{2^n} - \zeta^{2^m} \\ &= \left(X^{2^{n-1}} - \zeta^{2^{m-1}}\right)\left(X^{2^{n-1}} + \zeta^{2^{m-1}}\right) \\ &= \left(X^{2^{n-1}} - \zeta^{2^{m-1}}\right)\left(X^{2^{n-1}} - \zeta^{2^m+2^{m-1}}\right) \\ &= \cdots \end{aligned}$$

This process can be recursively iterated for sufficiently high n and m, which ultimately leads to the following isomorphisms thanks to the CRT [Ber01]. As such, for any ring R that contains a power-of-two primitive root of unity, the following holds.

$$\begin{aligned} \mathsf{R}[X]/(X^{2^n} + 1) &\cong \mathsf{R}[X]/(X^{2^{n-1}} - \zeta^{2^{m-1}}) \times \mathsf{R}[X]/(X^{2^{n-1}} - \zeta^{2^m+2^{m-1}}) \\ &\cong \quad \vdots \\ &\cong \mathsf{R}[X]/(X^{2^{n-k}} - \zeta^{2^{m-k}}) \times \mathsf{R}[X]/(X^{2^{n-k}} - \zeta^{2^m+2^{m-k}}) \times \cdots \end{aligned}$$

Modular polynomial multiplication over $\mathbb{Z}_q[X]/(X^n + 1)$ can thus be reduced to component-wise multiplications in "smaller" polynomial rings.

We first started by proving in Rocq the CRT for polynomials:

Theorem 1. *Let $p_1, p_2 \in \mathbb{K}[X]$ be two coprime polynomials with coefficients over a field $\mathbb{K}$. Then, CRT_2, as defined below, is a ring isomorphism.*

$$\begin{aligned} \mathsf{CRT}_2 : \mathbb{K}[X]/(p_1p_2) &\longrightarrow \mathbb{K}[X]/p_1 \times \mathbb{K}[X]/p_2 \\ p \bmod p_1p_2 &\longmapsto (p \bmod p_1, p \bmod p_2) \end{aligned}$$

In Rocq, we define the inverse iCRT_2 using the standard extended euclidean division algorithm, and show that they are both indeed inverse of each other, and both preserve ring operations.

We can specialize the CRT to our needs. Specifically, for $a \in \mathbb{K} \setminus \{0\}$, $X^n - a$ and $X^n + a$ are coprime when $n > 0$, thus, $\mathbb{K}[X]/(X^{2n} - a^2) \cong \mathbb{K}[X]/(X^n - a) \times \mathbb{K}[X]/(X^n + a)$ by CRT.

As explained earlier, we want to define the NTT to have a form close to

$$\mathsf{NTT}(p) = (p \bmod X^k - a_0, \ldots, p \bmod X^k - a_n)$$

We thus need to be able to define the $a_o, \ldots, a_n$. We thus first define a recursive function $\texttt{decompose} : \mathbb{N} \times \mathbb{N} \to \texttt{list}\ \mathbb{N}$ as follows.[1]

$$\texttt{decompose}(m, r, l) = \begin{cases} [l] & \text{if } r = 0 \\ \texttt{decompose}(m, r-1, \frac{l}{2}) +\!\!+ \texttt{decompose}(m, r-1, 2^m + \frac{l}{2}) & \text{if } r > 0 \end{cases}$$

We can easily prove by induction that $\mathsf{decompose}(m, r, l)$ is a list of 2^r integers. Furthermore, if we assume that we have a field $\mathbb{Z}_q$, $m \in \mathbb{N}$ and $\zeta \in \mathbb{Z}_q$ such that $\zeta^{2^m} = -1$,

[1] We use $+\!\!+$ to denote list concatenation. Moreover, we use $\frac{l}{2}$ to denote the quotient of the Euclidean division of l by 2, i.e., the unique integer q such that $l = 2q + l \bmod 2$.

we can also prove the following identity by induction on r that for all $n, l \in \mathbb{N}$ such that $r \leq \min(m, n)$ and $l \bmod 2^r = 0$, then

$$X^{2^n} - \zeta^l = \prod_{k \in \texttt{decompose}(m,r,l)} \left(X^{2^{n-r}} - \zeta^k\right)$$

This is immediate for $r = 0$ since $\texttt{decompose(m, 0, l)} = [l]$. If $r = r' + 1$, then we have that

$$\begin{aligned}
X^{2^n} - \zeta^l &= \left(X^{2^{n-1}} - \zeta^{\frac{l}{2}}\right)\left(X^{2^{n-1}} - \zeta^{2^m + \frac{l}{2}}\right) \\
&= \left(\prod_{k \in \texttt{decompose}(m,r',\frac{l}{2})} \left(X^{2^{n-1-r'}} - \zeta^k\right)\right)\left(\prod_{k \in \texttt{decompose}(m,r',2^m+\frac{l}{2})} \left(X^{2^{n-1-r'}} - \zeta^k\right)\right) \\
&= \left(\prod_{k \in \texttt{decompose}(m,r',\frac{l}{2}) \mathbin{+\!\!+} \texttt{decompose}(m,r',2^m+\frac{l}{2})} \left(X^{2^{n-(r'+1)}} - \zeta^k\right)\right) \\
&= \prod_{k \in \texttt{decompose}(m,r,l)} \left(X^{2^{n-r}} - \zeta^k\right)
\end{aligned}$$

To use the induction hypothesis, we rely only on that since $l \bmod 2^r = 0$ and $r = r' + 1$, we can conclude that $\frac{l}{2}$ is an exact integer, and $\frac{l}{2} \bmod 2^{r'} = 0$. Similarly, since $r' < r \leq min(m, n) \leq m$, we also have that $2^m \bmod 2^{r'} = 0$, thus $2^m + \frac{l}{2} \bmod 2^{r'} = 0$. Finally, we conclude by associativity of multiplication in the polynomial ring, as well as the definition of `decompose`.

By defining $\texttt{decomposition}(r, m, n, l) = \texttt{map}\ (\lambda k \mapsto X^{2^{n-r}} - \zeta^k)\ \texttt{decompose}(m, r, l)$ and by plugging $l = 2^m$ in the above identity, we have that for all $r \leq \min(m, n)$:

$$\begin{aligned}
X^{2^n} + 1 = X^{2^n} - \zeta^{2^m} &= \prod_{P(X) \in \texttt{decomposition(0,m,n,2^m)}} P(X) \\
&= \prod_{P(X) \in \texttt{decomposition(r,m,n,2^m)}} P(X)
\end{aligned}$$

We can thus define the NTT as follows

$$\texttt{ntt}(r, m, n, p) = \texttt{map}\ (\lambda\ q \mapsto p \bmod q)\ \texttt{decomposition}(r, m, n, 2^m)$$

Unfortunately, while this is suitable as a specification of the NTT, the computational content is not clearly apparent, which we need to derive code from it. Fortunately, we can recover it while proving that it is indeed a ring isomorphism.

As such, we define recursively auxiliary functions

$$\texttt{ntt}'(r, m, n, l) : \prod_{P(X) \in \texttt{decomposition}(0,m,n,l)} \frac{\mathbb{Z}_q[X]}{P(X)} \to \prod_{P(X) \in \texttt{decomposition}(r,m,n,l)} \frac{\mathbb{Z}_q[X]}{P(X)}$$

$$\texttt{intt}'(r, m, n, l) : \prod_{P(X) \in \texttt{decomposition}(r,m,n,l)} \frac{\mathbb{Z}_q[X]}{P(X)} \to \prod_{P(X) \in \texttt{decomposition}(0,m,n,l)} \frac{\mathbb{Z}_q[X]}{P(X)}$$

as follows.[2]

[2] ++ denotes a dependently-typed list concatenation this time, which we elide the explanation for simplicity. We use `split` to split a list into its two halves.

$$\mathtt{ntt}'(r,m,n,l,\rho) = \begin{cases} \rho & \text{if } r = 0 \\ \mathtt{let}\ (\rho_1,\rho_2) = \mathsf{CRT}_2(\rho)\ \mathtt{in} & \text{if } r > 0 \\ \quad \mathtt{ntt}'(r-1,m,n-1,\frac{l}{2},\rho_1) +\!\!+ & \\ \quad \mathtt{ntt}'(r-1,m,n-1,2^m+\frac{l}{2},\rho_2) & \end{cases}$$

$$\mathtt{intt}'(r,m,n,l,\rho) = \begin{cases} \rho & \text{if } r = 0 \\ \mathtt{let}\ (\rho_1,\rho_2) = \mathtt{split}(\rho)\ \mathtt{in} & \text{if } r > 0 \\ \quad \mathsf{iCRT}_2(\mathtt{intt}'(r-1,m,n-1,\frac{l}{2},\rho_1), & \\ \qquad\qquad \mathtt{intt}'(r-1,m,n-1,2^m+\frac{l}{2},\rho_2)) & \end{cases}$$

It is straightforward to prove by induction on r that for all $n, l \in \mathbb{N}$ such that $r \leq \min(m,n)$ and $l \bmod 2^r = 0$, then $\mathtt{ntt}'(r,m,n,l)$ $\mathtt{intt}'(r,m,n,l)$ are inverse of each other and are isomorphisms as compositions of the CRT_2 (resp., iCRT_2) isomorphism and recursive calls that are isomorphisms by induction hypothesis using similar reasoning as before. Furthermore, using similar reasoning as well as the fact that $(p \bmod ab) \bmod a = p \bmod a$, we can finally prove that

$$\mathtt{ntt}(r,m,n,p) = \mathtt{ntt}'(r,m,n,2^m,p)$$

2.3 Correctness of our Specification

Our specification of the NTT is "algebraic" which differs from what may appear in official standards. It it thus imperative to make sure that our specification matches.

Indeed, the codomain of our NTT is currently specified as:[3]

$$\prod_{k \in \mathtt{decompose}(m,\min(m,n),2^m)} \frac{\mathbb{Z}_q[X]}{X^{2^{n-\min(m,n)}} - \zeta^k}$$

However, the NIST standards specify the codomain of the NTT using bit-reversal for the powers of ζ, e.g., for ML-KEM [Nat24b, Eq. (4.11)], the codomain is defined as

$$\prod_{i=0}^{127} \frac{\mathbb{Z}_q[X]}{X^2 - \zeta^{2\mathsf{BitRev}_7(i)+1}}$$

We thus need to make sure that our specification matches with the ones used in practice in order for it to be useful. We have thus proved that for all m, $\mathsf{decompose}(m,m,2^m)$ is equal to the list of $2\mathsf{BitRev}_m(i)+1$ for $0 \leq i < 2^m$. Finally, for $n \geq m$, we have that

$$\mathtt{ntt}(m,m,n,p) = (p \bmod X^{2^{n-m}} - \zeta^{2\mathsf{BitRev}_m(0)+1}, \ldots, p \bmod X^{2^{n-m}} - \zeta^{2\mathsf{BitRev}_m(2^m-1)+1})$$

3 Lowering of the Specification

Now that we have a specification of NTT, we could start on generating verified C code out of it. However, directly generating code from this specification is a bit difficult as it is recursive and far from the imperative code we want to synthetize. We thus write lower-level code which we show to *refine* the specification.

[3] By unfolding `decomposition` and using $r = \min(m,n)$ and $l = 2^m$.

3.1 Low-level Gallina code

3.1.1 Changing the Representation

An issue with our specification is that we consider *abstract* polynomials. As the C code we want to synthesize will manipulate arrays of coefficients, we bridge the gap by first proving that any polynomial of degree lower than n can be uniquely represented by a list of its first n coefficients. Formally, this corresponds to proving that when $\mathsf{deg}(P(X)) = n$, then the following are isomorphic[4]

$$\frac{\mathbb{Z}_q[X]}{P(X)} \cong (\mathbb{Z}_q)^n$$

As explained in the NIST standards, "the choice of data structure for the inputs and outputs of NTT and NTT^{-1} are length-n arrays of integers" [Nat24b, §4.3]. However, our specification of the NTT and its inverse handle lists of polynomials. We thus show that we can "flatten" each layer of the NTT to a list of 2^n coefficients. This is immediate by induction as each layer is composed of 2^i polynomials of degree lower than 2^{n-i}.

$$\prod_{k \in \mathtt{decompose}(m,i,l)} \frac{\mathbb{Z}_q[X]}{X^{2^{n-i}} - \zeta^k} \cong \prod_{k=1}^{2^i} (\mathbb{Z}_q)^{2^{n-i}} \cong (\mathbb{Z}_q)^{2^n}$$

3.1.2 From Recursive to Iterative

Our NTT (specifically $\mathtt{ntt}'$ and $\mathtt{intt}'$) are currently defined recursively, which is far from the imperative loops we want to synthetize.

We first show that $\mathtt{ntt}'$ satisfies the following recursive equation[5]

$$\mathtt{ntt}'(r+1, m, n, l, p) = \mathtt{flat\_map}\ (\lambda p' \mapsto \mathsf{CRT}_2(p'))\ \mathtt{ntt}'(r, m, n, l, p)$$

We thus have that $\mathtt{ntt}'(r, m, n, l, p)$ successively applies a "state-transforming" function $f = \lambda\rho \mapsto (\mathtt{flat\_map}\ (\lambda p' \mapsto \mathsf{CRT}_2(p'))\ \rho)$ which can conceptually be represented as a loop.

Indeed, functionally, a for-loop `for (i = a; i < b; i++) { c }` can be represented as $\mathsf{fold}\ (\lambda(i, s) \mapsto f(i, s))\ s_0\ [a; \ldots; b-1]$ where f models the effects of command `c` on some state for which s_0 is the initial state.[6]

We can thus prove the following:[7]

$$\mathtt{ntt}'(r, m, n, l, p) = \mathsf{fold}\ f\ p\ [0; \ldots; r-1]$$

Similarly, as $\mathtt{flat\_map}\ (\lambda p' \mapsto \mathsf{CRT}_2(p'))\ \rho$ computes in-place, it can also be roughly represented as a loop:

$$\mathtt{flat\_map}\ (\lambda p' \mapsto \mathsf{CRT}_2(p'))\ \rho = \mathsf{fold}\ \mathsf{CRT}_2\ \rho\ [0; \ldots; \mathsf{length}(\rho) - 1]$$

Finally, CRT_2 can also be represented as a loop as we will see next. This explains why the algorithm for the NTT is defined as 3 nested loops in the NIST standards for ML-KEM [Nat24b] or ML-DSA [Nat24a].

[4]The addition for ring $(\mathbb{Z}_q)^n$ is the expected coefficient-wise addition, though multiplication is dependent on $P(X)$ which we elide for simplicity.

[5]$\mathtt{flat\_map}\ f\ l$ applies f to each element of the list l, and then flattens the resulting list. For instance, $\mathtt{flat\_map}\ (\lambda k \mapsto [2k; 2k+1])\ [0; 1; 2] = \mathtt{flatten}\ [[0; 1]; [2; 3]; [4; 5]] = [0; 1; 2; 3; 4; 5]$.

[6]We define $\mathsf{fold}\ f\ s_0\ l$ as s_0 when l is the empty list, $\mathsf{fold}\ f\ (f(i, s_0))\ l'$ when $l = i :: l'$.

[7]CRT_2 depends implicitly on the iteration number i which we elide for conciseness. We also define $[0; \ldots; r-1]$ as the empty list when $r = 0$.

3.2 Optimizations

We now present some optimizations that can be made independent of the NTT parameters. For instance, as seen Section 2.2, our specification calls a general procedure to compute a polynomial modulus. The specific shape of these moduli $X^{2n} - a^2$ can be exploited to have simpler and more efficient procedures to compute polynomial moduli as we explain next.

3.2.1 Specialized Polynomial Modular Operations

Indeed, in that context, for $p \in \mathbb{K}[X]/(X^{2n} - a^2)$, there exists $(c_i)_{0 \leq i < 2n} \in \mathbb{K}^{2n}$ such that $p = \sum_{i=0}^{2n-1} c_i X^i$, and we can easily compute $p \bmod (X^n - a)$ and $p \bmod (X^n + a)$:

$$\begin{aligned}
p &= \sum_{i=0}^{2n-1} c_i X^i \\
&= \left(\sum_{i=0}^{n-1} c_i X^i\right) + \left(X^n \sum_{i=0}^{n-1} c_{i+n} X^i\right) \\
&= \left(\sum_{i=0}^{n-1} (c_i - a c_{i+n}) X^i\right) + \left((X^n + a) \sum_{i=0}^{n-1} c_{i+n} X^i\right) \\
&= \left(\sum_{i=0}^{n-1} (c_i + a c_{i+n}) X^i\right) + \left((X^n - a) \sum_{i=0}^{n-1} c_{i+n} X^i\right)
\end{aligned}$$

Therefore,

$$\begin{aligned}
p \bmod (X^n - a) &= \sum_{i=0}^{n-1} (c_i + a c_{i+n}) X^i \\
p \bmod (X^n + a) &= \sum_{i=0}^{n-1} (c_i - a c_{i+n}) X^i
\end{aligned}$$

Inversely, if we let $u = p \bmod (X^n - a)$ and $v = p \bmod (X^n + a)$, then we have that:

$$\begin{aligned}
\frac{u+v}{2} &= \sum_{i=0}^{n-1} c_i X^i \\
\frac{u-v}{2a} &= \sum_{i=0}^{n-1} c_{i+n} X^i
\end{aligned}$$

Therefore,

$$p = \frac{1}{2}\left(u + v + \frac{u-v}{a} X^n\right)$$

Furthermore, this translates well to simple array manipulation as it shows that the i-th coefficient of u and v only depends on the i-th and $i+n$-th coefficients of p, as well as a known value a. Inversely, to recover p from u and v, the i-th and $i+n$-th coefficients of p only depend on both the i-th coefficients of u and v, as well as a known value $\frac{1}{2a}$. This shows how CRT_2 can be easily translated as a loop.

3.2.2 Delayed Multiplication

As we just explained, during the inverse NTT, one will need to compute $p = \frac{1}{2}\left(u + v + \frac{u-v}{a} X^n\right)$ which includes a modular multiplication by $\frac{1}{2}$. Rather than doing this multiplication for each layer of the transformation, it is possible to do them all at once at the end by performing a modular multiplication by 2^{-r} instead. This is an optimization presented in the NIST standards (e.g., [Nat24b, Alg. 10, l. 14], [Nat24a, Alg. 42, l. 21-24]), though not explained.

To show that this is correct, we first prove that not doing the multiplications up to layer r is equivalent to returning the expected result scaled up by 2^r.

$$\texttt{intt\_no\_mul}'(r, m, n, l, \rho) = \texttt{map}\ (\lambda p \mapsto 2^r p)\ \texttt{intt}'(r, m, n, l, \rho)$$

It thus immediately ensues that we can precompute $2^{-\min(m,n)} \bmod q$ and modularly multiply all the coefficients by it at the end after $\min(m, n)$ layers to scale back to the expected result.

3.2.3 Precomputed Array of ζs

In Section 2.2, we wrote that

$$\mathbb{Z}_q[X]/(X^{2^n} + 1) \cong \prod_{i \in \texttt{decompose}(m,r,2^m)} \mathbb{Z}_q[X]/(X^{2^{n-r}} - \zeta^i)$$

By definition of the list $\texttt{decompose}(m, r, l)$, for $r > 0$, if i is at an even index $2k$ of $\texttt{decompose}(m, r, l)$, and i' is at index $2k + 1$, we have that $\zeta^{i'} = \zeta^{2^m+i} = -\zeta^i$.

We can thus write for $0 < r \leq \min(m, n)$ that

$$\mathbb{Z}_q[X]/(X^{2^n}+1) \cong \prod_{i=(\texttt{decompose}(m,r,2^m))[2k]}^{0\leq k<2^{r-1}} \Big(\mathbb{Z}_q[X]/(X^{2^{n-r}} - \zeta^i)\Big) \times \Big(\mathbb{Z}_q[X]/(X^{2^{n-r}} + \zeta^i)\Big)$$

Thus, as mentioned in the NIST standards, one can precompute the ζ^i used in computing

$$p \bmod (X^{2^{n-r}} - \zeta^i) = \sum_{k=0}^{2^{n-r}-1} (c_k + \zeta^i c_{k+2^{n-r}}) X^k$$
$$p \bmod (X^{2^{n-r}} + \zeta^i) = \sum_{k=0}^{2^{n-r}-1} (c_k - \zeta^i c_{k+2^{n-r}}) X^k$$

This results in precomputing an array of $\sum_{i=1}^{\min(m,n)} 2^{i-1} = 2^{\min(m,n)} - 1$ values.

Inversely, as explained in Section 2.2, to recover p from $p \bmod (X^{2^{n-r}} - \zeta^i)$ and $p \bmod (X^{2^{n-r}} + \zeta^i)$, one must perform a modular multiplication by $\frac{1}{\zeta^i}$.

It is possible to similarly precompute an array of all these values, though that would be a waste of space. Indeed, one can prove by induction on $r \leq m$ that, for all $k < 2^r$, $(\texttt{decompose}(m, r, 2^m))[k] + (\texttt{decompose}(m, r, 2^m))[2^r - 1 - k] = 2 * 2^m$.

It is immediate for $r = 0$ since necessarily $k = 0$, and thus $2^m + 2^m = 2 * 2^m$. In the inductive case, for $k < 2^{r+1}$, let $k' = \lfloor \frac{k}{2} \rfloor$, then $k' < 2^r$. We also have that either $k = 2k'$ or $k = 2k' + 1$.

In the case $k = 2k'$, we can prove that

$$(\texttt{decompose}(m, r+1, 2^m))[k] = \frac{(\texttt{decompose}(m, r, 2^m))[k']}{2}$$
$$(\texttt{decompose}(m, r+1, 2^m))[2^{r+1} - 1 - k] = \left(2^m + \frac{(\texttt{decompose}(m, r, 2^m))[2^r - 1 - k']}{2}\right)$$

Thus, by using the induction hypothesis, we have that

$$\begin{aligned}(\texttt{decompose}(m, r+1, 2^m))[k] + (\texttt{decompose}(m, r+1, 2^m))[2^{r+1} - 1 - k] &= 2^m + \frac{2 * 2^m}{2} \\ &= 2 * 2^m\end{aligned}$$

We can use a similar reasoning the second case when $k = 2k' + 1$, and the property is thus proved by induction.

We can thus conclude that if we let $i = (\texttt{decompose}(m, r, 2^m))[2k]$ for $r > 0$ such that ζ^i is already part of the array of precomputed values, then let $i_1 = (\texttt{decompose}(m, r, 2^m))[2^r - 1 - 2k]$, and we have that $i + i_1 = 2 * 2^m$ per our lemma. However, since $2^r - 1 - 2k$ is odd, then ζ^{i_1} is not in our precomputed array. However, we have that $2^r - 1 - 2k - 1$ is even, and thus for $i_2 = (\texttt{decompose}(m, r, 2^m))[2^r - 1 - 2k]$, we have that $\zeta^{i_1} = -\zeta^{i_2}$ and ζ^{i_2} is in the precomputed array.

Finally, since $\zeta^{2*2^m} = \left(\zeta^{2^m}\right)^2 = (-1)^2 = 1$, we have that

$$\begin{aligned}1 &= \zeta^i * \zeta^{i_1} \\ &= \zeta^i * (-\zeta^{i_2})\end{aligned}$$

Hence, $\frac{1}{\zeta^i} = -\zeta^{i_2}$ and the precomputed array used for the forward NTT can thus be also used for the inverse NTT.

3.2.4 Delayed Modular Reduction

So far, we have shown that the NTT is morally equivalent to computing the following pseudo-code.

```
ntt(r,n,p) :=
    l = 0;
    len = 2^n;
    for (a = 0; a < r; a++) {
      old_len = len;
      len = len >> 1;
      start = 0;
      for (b = 0; b < (1 << a); b++) {
        m = m + 1;
        z = zetas[l];
        j = start;
        for (j = start; j < start + len; j++) {
          t = z * p[j + len];
          p[j + len] = p[j] - t;
          p[j] = p[j] + t;
        };
        start = start + old_len;
      }
    }
  }
```

The innermost loop of the NTT implements a so-called "butterfly" operation:

$$\begin{aligned}y[i] &= x[i] + z * x[i+n] \\ y[i+n] &= x[i] - z * x[i+n]\end{aligned}$$

These operations represent modular arithmetic in a field $\mathbb{Z}_q$. When *implemented as code*, it is necessary to choose a representation for $\mathbb{Z}_q$. For instance, the reference implementation for Kyber [SAB+20] has chosen a signed representation (i.e., the representative of $x \in \mathbb{Z}_q$ is in the interval $[-(q-1)/2; (q-1)/2]$ for q an odd prime), whereas elliptic curve cryptography has usually chosen an unsigned representation (i.e., the representative of $x \in \mathbb{Z}_q$ is in the interval $[0; q-1]$). Let $b_q = \max\{|x| \mid x \in \mathbb{Z}_q\}$, then when q is an odd prime, $b_q = (q-1)/2$ when using a signed representation, $b_q = q - 1$ when using an unsigned representation.

Modular reduction is usually implemented by a function `reduce` such that there exists a bound b and for all $|x| \leq b$, $\texttt{reduce}(x) = x \bmod q$ with $|\texttt{reduce}(x)| \leq b_q$. As one needs to at least be able to compute the multiplication of two field elements, we usually have that $b_q * b_q \leq b$. The innermost loop of the NTT is thus implemented as follows (ignoring big integer issues for now).

```
t = reduce(z * p[j + len]);
p[j + len] = reduce(p[j] - t); // or reduce(p[j] + (q - t))
p[j] = reduce(p[j] + t);
```

Let $k = \lfloor b/b_q^2 \rfloor$, we have that $1 \leq k$. We can show that when implementing the NTT, it is possible to not reduce the additions and substractions in the butterfly operation for $k - 1$ layer iterations, but only reduce at the k-th iteration. This is because after $k - 1$ iterations, the coefficients in the array can be bounded by kb_q, and thus $z * p[j + len]$ can be bounded by $kb_q \times b_q \leq b$ by definition.

Let $r = r_k \times k + r'_k$ be the euclidean division of r by k, then the outermost loop of the NTT can be replaced by r_k loops each doing $k - 1$ iterations without reduction, and a last k-th iteration with modular reduction. If $r'_k > 0$, these iterations can be replaced with $r'_k - 1$ iterations without reduction and the r'_k-th iteration with reduction.

Similar reasoning can be made for the inverse NTT, though we have not yet formalized it in Rocq. Furthermore, we have only formalized this optimization using the unsigned representation as Fiat-Crypto has only implemented this representation. Our proof should be easily generalizable once Fiat-Crypto implements signed representation, but we consider it orthogonal to the work we present here.

As a side-note, it is unclear whether this optimization is always beneficial. For instance, in the case of ML-KEM, the prime are small compared to the integer type (e.g., `int16_t`) used to represent the field element, which allows to avoid all modular additions and subtractions until the very last layer. On the other hand, if $q = 2^{255} - 19$ (e.g., for Prio3 when using the largest parameters) and a "saturated" representation is chosen where field elements are represented by four 64-bits limbs, it seems necessary that reduction happens at each iteration as $3q > 2^{256}$. However, if an "unsaturated" representation with five 64-bits limbs is used, it seems likely that delaying reduction could be beneficial. We have not yet explored this.

4 Formally Verified Code Synthesis

We now explain how we derived formally verified C implementations of the NTT using Fiat-Crypto, which we present first.

4.1 Background

In this section, we present some background on Fiat-Crypto [EPG+19] and Bedrock2 [EGC+21, EPJ+24] which we use to synthesize formally verified C implementations of the NTT.

4.1.1 Fiat-Crypto

Fiat-Crypto is a framework embedded inside of the Rocq proof assistant for synthesizing correct-by-construction code for cryptography primitives. It has been mainly used to produce high-speed and high-assurance implementations of cryptographic arithmetic, e.g., modular arithmetic for large finite fields that is necessary for implementing elliptic curve cryptography. Code produced by Fiat-Crypto is now used in multiple popular cryptographic libraries such as BoringSSL and NSS, as well as their corresponding browsers Google Chrome and Mozilla Firefox.

From a high-level point of view, Fiat-Crypto can be split into two parts: a formal library of mathematical facts and algorithms related to cryptography, and a synthesis pipeline that specializes those algorithms and outputs efficient code.

As Fiat-Crypto has been extensively used for elliptic curve cryptography, its mathematical library already contains general facts about modular arithmetic, Barrett and Montgomery reductions, multi-limbed arithmetic, etc, that may be also useful for implementing lattice-based cryptography. This is what we build upon to formalize our NTT.

Originally, Fiat-Crypto used a formally proven correct partial evaluator [GEP+22, GEP+24] to translate the mathematical specifications written in the Gallina language of Rocq into a low-level Fiat-Crypto intermediate representation which were then translated to C through an unverified backend. Facilities were recently added to output verified C code through a Bedrock2 backend that we present below [EPJ+24, §3.6]. This has improved confidence in the correctness of the generated code.

4.1.2 Bedrock2

Bedrock2 is also a framework embedded within the Rocq proof assistant that formally specifies a C-like source language for low-level programming and provides a formally verified compiler for this language down to RISC-V machine code. The Bedrock2 source language has fairly simple semantics, especially compared to C which is notoriously intricate. It is completely untyped and has a single type of value: machine words (i.e., 32 or 64-bits integers). Interpretation of these values as pointers or integers depends only on the operators they are used with, e.g., a load/store operation or a (un)signed comparison. The only control-flow structures allowed are while loops and conditional branchings. Thanks to simplicity, it is possible to pretty-print Bedrock2 programs to C code.

In order to reason about programs written in the language, Bedrock2 provides a program logic based on separation logic. For instance, the following is a specification of a function that computes the field addition of its inputs.

```
fnspec ∃ f, "felem_add" (x y z: word)
                 / (a b c: F) (R: _ -> Prop)
                 ~> (res: word),
      { requires t m k :=
          (FElem x a * FElem y b * FElem z c * R)%sep m;
        ensures T M K :=
          T = t ∧ res = word.zero ∧ K = f x y z k ∧
          (FElem x a * FElem y b * FElem z (a + b)%F * R)%sep M
      }
```

More specifically, the above specification states that

- there exists a function `f` such that, when calling the function `felem_add` provided with three machine words `x`, `y`, `z` as inputs,
- such that they represent locations in memory state `m` where the machine word representations of finite field elements `a`, `b`, `c` are respectively stored (this is the `FElem` predicate),
- then the function will return the machine word 0 (`res = word.zero`),

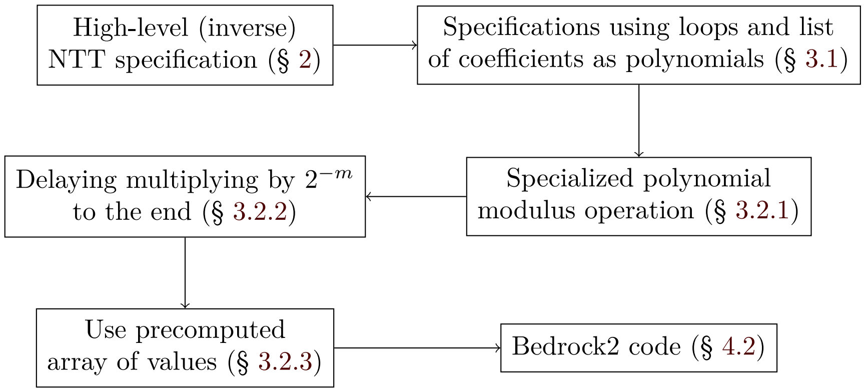

Figure 2: Description of the code synthesis process.

- no externally observable behavior has occurred (`T = t`),
- the leakage `K` produced by `felem_add` only depends on the inputs `x`, `y`, `z` and the previous leakage `k` (`K = f x y z k`),
- and memory has been modified into a state `M` such that the result of field addition of `a` and `b` (written as `(a + b)%F`) will now be stored at location `z`, while the rest of the memory is unchanged.

The `R` memory predicate describes how the rest of the memory that is of no importance to the function is shaped. It being unmodified between the `requires` and `ensures` is how we specify that the function does not modify it. The separating conjunction $*$ implies that the memory locations defined by `x`, `y` and `z` are separated, i.e., they cannot be aliases.

The leakage condition is the Bedrock2 way to specify that the function is cryptographically constant-time [CEC25]. Specifically, as the leakage function `f` is existentially quantified *before* the other parameters, it cannot depend on them. As such, this says that the only information potentially leaked is the memory locations of the field elements, but not the content itself. The leakage can also depend on public constant values, e.g., an offset in an array.

Note that this specification uses bits from Fiat-Crypto such as the notation for finite field addition `(_ + _)%F`, but does not describe how the code accomplishes it (e.g., naively or using a Barrett reduction). This is only needed for the *proof* that the code behaves as specified, which will allow us to be modular.

4.2 Code Synthesis

We now explain how we synthesize formally verified C code. Figure 2 shows the successive transformations we applied to the original high-level specifications to obtain specifications closer to the code we want to generate, as presented earlier. Delayed modular reduction (§ 3.2.4) has been formalized and proven correct at the specification level, though we have not used it to derived code yet. The methodology presented in the following would still apply however.

So far, our specifications have used mathematical integers, however our code will have to deal with machine words. As such, we use width to denote the machine word width, i.e., 32 or 64 bits.

For simplicity, we assume here that the NTT operates over a field $\mathbb{Z}_q$ such that $\lceil \log_2(q) \rceil < \mathsf{width}$.[8] Moreover, since the polynomial coefficients are stored in memory, we

[8] We have also proved correct a version of the NTT where multi-limb arithmetic is used, which we elide for space.

need to ensure that they are addressable, hence, if there are 2^n coefficients, then we need that $n < \mathsf{width}$.

We further assume to already have access to constant-time implementations of finite field operations for $\mathbb{Z}_q$. This allows us to be modular and specialize code generation at the end to the modular arithmetic we wish to use, e.g., using Barrett or Montgomery reduction, etc.

```
Definition spec_of_binop {name: String.string}
           (model: F -> F -> F): spec_of name :=
  fnspec! ∃ f, name (x y: word) / (a b: F) ~> (res: word),
  { requires tr mem k :=
    feval x = Some a /\ feval y = Some b;
    ensures TR MEM K :=
    TR = tr /\ MEM = mem /\ K = f k /\
      feval res = Some (model a b)
  }.
Instance spec_of_add: spec_of add :=
  spec_of_binop F.add.
Instance spec_of_sub: spec_of sub :=
  spec_of_binop F.sub.
Instance spec_of_mul: spec_of mul :=
  spec_of_binop F.mul.
```

Specifically, we assume that we have access to functions with names `add`, `sub` and `mul` that correctly implements their respective specifications. For instance, we assume that `add` satisfies its specification `spec_of_add` which states that the functions takes 2 machine words `x` and `y` as inputs such that if they respectively represent finite field elements `a` and `b` (written as `feval x = Some a`), then the function returns a machine word `res` such that it represents the result of modular addition $\mathtt{a} + \mathtt{b}$, without modifying the memory state. The specification is parametric with regards to what definition of `feval` is used, this allows to use the same specification no matter the representation used for field elements, e.g., Montgomery representation, reduced unsigned representation, etc.

With those, we can now write a Bedrock2 function[9] that implements the NTT:

```
Definition br2_ntt :=
  func! (p) {
    m = ⟨0⟩;
    len = ⟨2^n⟩;
    while (⟨2^(n−min(n,m))⟩ < len) {
      old_len = len;
      len = len >> ⟨1⟩;
      start = ⟨0⟩;
      while (start < 2^n)) {
        m = m + ⟨1⟩;
        z = zetas[m];
        j = start;
        while (j < (start + len)) {
          x = load(p + (j + len));
          tmp = mul(z, x);
          y = load(p + j);
          x = sub(y, tmp);
          store(p + (j + len), x);
          x = add(y, tmp);
          store(p + j, x);
```

[9] This is slightly simplified for presentation purpose.

```
          j = j + ⟨1⟩
        };
        start = start + old_len
      }
    }
  }.
```

One can notice that this looks very similar to the pseudo-code provided by the NIST standards (e.g., [Nat24b, Alg. 9], [Nat24a, Alg. 41]), though we use while loops rather than for loops. The code should be rather straightforward to read and understand.

One peculiarity is that the integer values that appear in the code within $\langle\cdot\rangle$ are mathematical (unbounded) integers that must be statically computable and convertible to machine words during code generation. By computable, we mean that parameters must be instantiated, for instance, in the case of ML-KEM, we have $n = 8$ and $m = 7$, which will translate 2^n to `256` and $2^{n-\min(n,m)}$ to `2` during code generation. By convertible, we mean that the integers must be representable as word machines, and we thus need to prove that it is possible. This is easy for `0` or `1`, but this is also why we needed the assumption $n <$ width. Similarly, `zetas` is an array of precomputed values defined as in Section 3.2.3. These values are finite field elements, and will also be translated into machine words during code generation.

Finally, we prove that our Bedrock2 implementation satisfies the following specification.

```
Instance spec_of_ntt: spec_of ntt :=
  fnspec! ∃ f, ntt (p_ptr: word) / (p: list F) R,
  { requires tr mem k :=
      exists (P: list word),
        Forall2 (fun x y => feval y = Some x) p P /\
        mem =* (Bignums1 2^n p_ptr P) * R;
    ensures TR MEM K :=
      TR = tr /\
      K = f p_ptr k /\
      exists (P: list word),
        Forall2 (fun x y => feval y = Some x) (NTT_spec p) P /\
        MEM =* (Bignums1 2^n p_ptr P) * R }.
```

The specification states that the function takes a single machine word `p_ptr` as input and returns nothing. It requires that `p_ptr` be a pointer that points to an array of size 2^n containing a list of machine words `P` as specified by `Bignums1` 2^n `p_ptr P`.[10] `P` represents a list of finite field elements p (i.e., the input polynomial) as specified by `Forall2 (fun x y => feval y = Some x) p P` where `feval` evaluates a machine word into a field element. The function returns no value, but has side-effects. Memory is modified such that the array that `p_ptr` pointed to, now contains a representation of `NTT_spec p`, which is our mathematical specification of the NTT. This specification further shows that all computations are made in place as the predicate representing the rest of memory `R` is not modified between the pre and post-conditions. Note that there are no assumptions on `zetas` in the specification, this is because we also generate the corresponding array. The leakage produced is only dependent on `p_ptr`, meaning that the array is only accessed at predictible offsets, which entails that the function is constant-time.

The proof that our Bedrock2 implementation satisfies the specification is mostly straightforward as we already proved correct a specification of the NTT using functional loops (§ 3.1.2).

[10] In the more general case of multi-limb field arithmetic, we use instead `Bignums` k 2^n `p_ptr P` to state that `p_ptr` points to an array of $k2^n$ words where each chunk of k words represent one field element.

We similarly define and specify the Bedrock2 implementation of the inverse NTT as follows.

```
Definition br2_ntt_inverse :=
  func! (p) {
    m = ⟨2^min(n,m)⟩;
    len = ⟨2^(n−min(n,m))⟩;
    while (len < ⟨2^n⟩) {
      start = ⟨0⟩;
      old_len = len;
      len = len << ⟨1⟩;
      while (start < ⟨2^n⟩) {
        m = m - ⟨1⟩;
        z = zetas[m];
        j = start;
        while (j < start + old_len) {
          tmp = load(p + j);
          x = load(p + (j + old_len));
          y = add(tmp, x);
          store(p + j, y);
          x = sub(x, tmp);
          y = mul(z, x);
          store(p + (j + old_len), y);
          j = j + ⟨1⟩
        };
        start = start + len
      }
    };
    j = ⟨0⟩;
    while (j < ⟨2^n⟩) {
      x = load(p + j);
      x = mul(⟨F.inv 2^min(n,m)⟩, x);
      store(p + j, x);
      j = j + ⟨1⟩
    }
  }.
```

```
Instance spec_of_ntt_inverse: spec_of ntt_inverse :=
  fnspec! ∃ f, ntt_inverse (p_ptr: word) / (p: list F) R,
  { requires tr mem k :=
      exists (P: list word),
      Forall2 (fun x y => feval y = Some x) p P /\
      mem =* (Bignums1 2^n p_ptr P) * R;
    ensures TR MEM K :=
      TR = tr /\
      K = f p_ptr k /\
      exists (P: list word),
      Forall2 (fun x y => feval y = Some x)
              (NTT_inverse_spec p) P /\
      MEM =* (Bignums1 2^n p_ptr P) * R }.
```

Again, this follows closely the pseudo-code provided by the NIST standards. The specification is the same as for the forward NTT, except that the array at the end contains instead a representation of `NTT_inverse_spec p`, which is our specification of the inverse NTT. The nested loops compute the inverse direction of the decomposition in the forward NTT, except that the result is scaled up as explained in Section 3.2.2. The bottom loop thus scales back the coefficients by the modular inverse of $2^{\min(n,m)}$. As explained earlier, the value ⟨`F.inv` $2^{\min(n,m)}$⟩ will be computed during code generation.

We proved that our implementations of the NTT are constant-time assuming that the functions implementing finite field arithmetic operators are also constant-time. While we do not detail it here, we formalized the signed representation for finite field arithmetic that is popular in implementations using small primes as in ML-KEM or ML-DSA. We proved and derived constant-time implementations using refined Barrett reduction [BHK+22] and signed Montgomery reduction [Hwa24]. We also proved and derived implementations using the usual unsigned representation from Fiat-Crypto.

5 Generating the Implementations

As explained in Section 4, we can generate C implementations of the forward and inverse NTT given that some parameters are provided. In this section, we first explain what parameters we need to synthesize C code. We then compare our formally verified implementations with some handwritten implementations and explain the differences.

5.1 Implementation Parameters

In order to generate C code, we need to instantiate the following parameters. We describe the process when using “small” primes, the process is slightly different for multi-limb field arithmetic.

1. width: we need to indicate the width of machine words for the machine we consider;
2. q: the size of the considered finite field $\mathbb{Z}_q$;
   (a) $\lceil \log_2(q) \rceil < \mathsf{width}$: as explained in Section 4, for the implementation to be correct, we need that $\lceil \log_2(q) \rceil < \mathsf{width}$ which can now be easily proved by computation;
   (b) q prime: this is also necessary to prove for the correctness of the implementation.
3. n: integer such that $\mathbb{Z}_q[X]/(X^{2^n} + 1)$ is the domain of the NTT;
   (a) $n < \mathsf{width}$: as explained in Section 4, we need 2^n to be convertible;
4. ζ: an element in $\mathbb{Z}_q$;
5. m: an integer;
   (a) $\zeta^{2^m} = -1 \bmod q$: this is necessary for correctness and easy to prove by computation.
6. ζs: an array of all $\zeta^0, \zeta^1, \ldots, \zeta^{2^m}$. This is necessary to implement the optimization explained in Section 3.2.3;
7. c: an integer such that it is the multiplicative inverse of $2^{\min(n,m)}$ in $\mathbb{Z}_q$, necessary for the optimization explained in Section 3.2.2.

Finally, as explained in Section 4, we need to choose how field arithmetic is implemented. Our Bedrock2 formalization currently only supports both signed and unsigned Barrett and Montgomery reductions.

5.2 Generated C Implementations

Using the presented machinery, we have generated C implementations for multiple PQC shemes using the parameters indicated in the table below. This shows that our formalization is general enough to model "incomplete" decompositions, e.g., ML-KEM, or high number of layers, e.g., 10 for Falcon-1024.

Algorithm	n	m	q	ζ
ML-KEM [Nat24b]	8	7	3329	17
ML-DSA [Nat24a]	8	8	8380417	1753
Falcon-512 [PFH+20]	9	10	12289	7
Falcon-1024 [PFH+20]	10	10	12289	7

These implementations were generated targeting a 64-bits machine and using unsigned representations for simplicity. We tested that they behaved as expected by checking that roundtrips (NTT followed by an inverse NTT) over randomly generated polynomials return the original polynomial.

We have also generated a verified implementation of the forward NTT for Prio3 using the Field128 parameters [BCPS25] to try out the code generation when using multi-limb field arithmetic. Unlike the previous examples, we do not output a precomputed array of twiddle factors, but expect a pointer to such an array. This is because the code generation facility already started to struggle computing the array for Falcon, which suggests that we are reaching the limits of Bedrock2 when $\min(m, n) \geq 10$ (Prio3's NTT may go up to 20 layers).

To evaluate our implementations, we modified PQClean [KSSW22]'s implementation of ML-KEM-512 to replace its NTT with ours and used the PQM4 [KPR+] framework to benchmark. We first modified the implementation to use 32-bits integers (`int32_t`) rather than 16-bits integers (`int16_t`) to represent the coefficients of the polynomials. As explained earlier, this is due to a limitation of Bedrock2 which only models machine-width integers. We verified that this has negligible impact on performance. We then replaced the forward NTT implementation with our verified one.[11]

Our implementation uses our verified implementations of signed Barrett and Montgomery reductions which are functionally equal to PQClean's implementations. We did not replace the inverse NTT as our implementation cannot be used as a drop-in replacement. This is because PQClean's inverse NTT leaves the coefficients in the Montgomery domain.[12] While we can use our specifications to verify this specific implementation of the inverse NTT, we prefered focusing on generic implementations.

We report below the average number of cycles taken by each stage of the scheme over 1000 runs on a Nucleo-L4R5ZI as reported by PQM4. The "clean" implementation corresponds to the reference PQClean implementation. "Fiat-Crypto" is our modified implementation using our generated code, while "m4fspeed" corresponds an implementation optimized for running on Cortex-M4.

Scheme	Implementation	Key Generation	Encapsulation	Decapsulation
ML-KEM-512	clean	512 161	574 798	706 145
ML-KEM-512	Fiat-Crypto	551 141	600 303	752 634
ML-KEM-512	m4fspeed	392 465	388 870	423 667

[11] We had to slightly modify manually our implementation as Bedrock2 models memory accesses in C similarly to the following: `x = _br_load(p+((br_word_t)4*(j+len)));` where `_br_load` is inline code that should be compiled away to a single load instruction. We replaced those by more idiomatic `x = ((int32_t*)p)[j+len]` which improved performance. See https://github.com/mit-plv/bedrock2/blob/04122ff2848f2f19de98dafe00caa1be4c370f89/bedrock2/src/bedrock2/ToCString.v#L36-L68 for details on `_br_load`.

[12] Base polynomial multiplication leaves coefficients in "inverse Montgomery domain" which is compensated by the inverse NTT.

Our implementation is slower than the PQClean implementation by roughly 5 to 7.5% depending on the stage in executing the algorithm. We believe this is because we do not implement the delayed reduction optimization presented in Section 3.2.4, which we confirmed by manually modifying our implementation to include the optimization. The performance were then similar to the PQClean implementation.The m4fspeed implementation uses hand-optimized assembly code with Plantard reduction and merged layers for the NTT [HZZ+22].

6 Related Work

We have already discussed other works on formally verifying the NTT in a proof assistant in Section 2, we review here related works on verified implementations of cryptography.

There have been multiple projects about producing high-assurance implementations of cryptography primitives. The main ones are Fiat-Crypto [EPG+19] that we discussed earlier, HACL* [ZBPB17] and Jasmin [ABB+17]. Fiat-Crypto can output formally verified C code, or machine code for the RISC-V architecture. Similarly, HACL* has verified C code, while Jasmin is a completely verified toolchain for the Jasmin language, and as such can produce highly-optimized verified assembly implementations for multiple architectures.

To the best of our knowledge, HACL* has currently no verified implementations of the NTT, though as explained earlier, a verified implementation of ML-KEM [AOB+24] in Jasmin exists.

In another line of work, Hwang et al. [HLS+22] use CryptoLine [PTWY18] to verify AVX2 and ARM Cortex-M4 implementations of NTT computations used by multiple NIST PQC candidates. While their approach applies directly to assembly implementations whereas we "only" bottom out at C level (or unoptimized RISC-V machine code), their TCB is also much bigger than ours as they rely on both Computer Algebra Systems (CAS) and SMT solvers. Moreover, they were only able to support very small primes as support for 32-bit computation was only added through work concurrent to ours [CLT+25]. Our formal specification of the NTT can be used to derive verified implementations even when multi-limb field arithmetic is necessary as we showed,[13] or to verify specific implementations.

7 Conclusion and Future Work

In this work, we showed how to synthesize *functionally correct*, *constant-time* and *memory-safe* C implementations of the NTT for multiple cryptographic schemes in a *foundational* way using the Rocq proof assistant. This work has taken around 7 person-months to develop and amounts to around 4300 lines of specification and 11k lines of proof in Rocq. This shows that current frameworks are advanced enough that it is now feasible to produce high-assurance implementations of critical cryptographic operations relatively quickly.

One immediate direction of future work is to incorporate the delayed reduction strategy presented in Section 3.2.4 into the code generation. Another possibility is to formalize Plantard arithmetic [Pla21, HZZ+22] which is used in the fastest implementations for some schemes.

We could also try to formalize the multiple ways of computing polynomial multiplications in lattice-based cryptosystems [Hwa24]. In particular, coefficient ring switching could be of interest to automatically derive a NTT for NTT-unfriendly rings [CHK+21] as used for instance in Saber [DKR+20].

[13] We point out that the issue of generating a large precomputed array of powers of ζ, as mentioned in the previous section, is a *limitation of the framework*, not the methodology we present.

As Fiat-Crypto was initially created for deriving verified implementations of finite field operations for elliptic curves, they only needed to model native machine words, and support for smaller integer types does not exist. It would thus be useful to add support for such integer types or target another backend such as the formal Rocq development of Jasmin [ABB+17] which would additionally provide an efficient formally verified compiler down to assembly for multiple architectures and further give access to SIMD instructions so more optimized implementations can be formalized.

References

[ABB+17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1807–1823. ACM Press, October / November 2017. doi:10.1145/3133956.3134078.

[ABB+23] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, Antoine Séré, and Pierre-Yves Strub. Formally verifying Kyber episode IV: Implementation correctness. *IACR TCHES*, 2023(3):164–193, 2023. doi:10.46586/tches.v2023.i3.164-193.

[AOB+24] José Bacelar Almeida, Santiago Arranz Olmos, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Cameron Low, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, and Pierre-Yves Strub. Formally verifying Kyber - episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part II*, volume 14921 of *LNCS*, pages 384–421. Springer, Cham, August 2024. doi:10.1007/978-3-031-68379-4_12.

[BBB+21] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. In *2021 IEEE Symposium on Security and Privacy*, pages 777–795. IEEE Computer Society Press, May 2021. doi:10.1109/SP40001.2021.00008.

[BCPS25] Richard Barnes, David Cook, Christopher Patton, and Phillipp Schoppmann. Verifiable Distributed Aggregation Functions. Internet-Draft draft-irtf-cfrg-vdaf-14, Internet Engineering Task Force, January 2025. Work in Progress. URL: https://datatracker.ietf.org/doc/draft-irtf-cfrg-vdaf/14/.

[Ber01] Daniel J. Bernstein. Multidigit multiplication for mathematicians. https://cr.yp.to/papers.html#m3, 2001.

[BGHZ11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 71–90. Springer, Berlin, Heidelberg, August 2011. doi:10.1007/978-3-642-22792-9_5.

[BHK+22] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Lorenz Panny, and Bo-Yin Yang. Efficient multiplication of somewhat small integers using number-theoretic transforms. In Chen-Mou Cheng and Mitsuaki Akiyama, editors, *Advances in Information and Computer Security - 17th International Workshop on Security, IWSEC 2022, Tokyo, Japan, August 31 - September 2, 2022, Proceedings*, volume 13504 of *Lecture Notes in Computer Science*, pages 3–23. Springer, 2022. URL: https://doi.org/10.1007/978-3-031-15255-9_1.

[Cap01] Venanzio Capretta. Certifying the fast fourier transform with coq. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001, Proceedings*, volume 2152 of *Lecture Notes in Computer Science*, pages 154–168. Springer, 2001. URL: https://doi.org/10.1007/3-540-44755-5_12.

[CB17] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In Aditya Akella and Jon Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 259–282. USENIX Association, 2017. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/corrigan-gibbs.

[CEC25] Owen Conoly, Andres Erbsen, and Adam Chlipala. Smooth, integrated proofs of cryptographic constant time for nondeterministic programs and compilers. *Proc. ACM Program. Lang.*, 9(PLDI), 2025. URL: https://doi.org/10.1145/3729318.

[CHK+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT multiplication for NTT-unfriendly rings. *IACR TCHES*, 2021(2):159–188, 2021. URL: https://tches.iacr.org/index.php/TCHES/article/view/8791, doi:10.46586/tches.v2021.i2.159-188.

[CLT+25] Chun-Ming Chiu, Jiaxiang Liu, Ming-Hsien Tsai, Xiaomu Shi, Bow-Yaw Wang, and Bo-Yin Yang. Algebraic linear analysis for number theoretic transform in lattice-based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(3):668–692, Jun. 2025. URL: https://tches.iacr.org/index.php/TCHES/article/view/12230, doi:10.46586/tches.v2025.i3.668-692.

[DKR+20] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. SABER. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[dMU21] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi:10.1007/978-3-030-79876-5_37.

[EGC+21] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. Integration verification across software and hardware for a simple embedded system. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 604–619. ACM, 2021. doi:10.1145/3453483.3454065.

[EPG+19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy*, pages 1202–1219. IEEE Computer Society Press, May 2019. doi:10.1109/SP.2019.00005.

[EPJ+24] Andres Erbsen, Jade Philipoom, Dustin Jamner, Ashley Lin, Samuel Gruetter, Clément Pit-Claudel, and Adam Chlipala. Foundational integration verification of a cryptographic server. *Proc. ACM Program. Lang.*, 8(PLDI):1704–1729, 2024. doi:10.1145/3656446.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009. doi:10.1145/1536414.1536440.

[GEP+22] Jason Gross, Andres Erbsen, Jade Philipoom, Miraya Poddar-Agrawal, and Adam Chlipala. Accelerating verified-compiler development with a verified rewriting engine. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 17:1–17:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: https://doi.org/10.4230/LIPIcs.ITP.2022.17, doi:10.4230/LIPICS.ITP.2022.17.

[GEP+24] Jason Gross, Andres Erbsen, Jade Philipoom, Rajashree Agrawal, and Adam Chlipala. Towards a scalable proof engine: A performant prototype rewriting primitive for coq. *J. Autom. Reason.*, 68(3):19, 2024. URL: https://doi.org/10.1007/s10817-024-09705-6, doi:10.1007/S10817-024-09705-6.

[Har09] John Harrison. HOL light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer, 2009. doi:10.1007/978-3-642-03359-9_4.

[HLS+22] Vincent Hwang, Jiaxiang Liu, Gregor Seiler, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verified NTT multiplications for NISTPQC KEM lattice finalists: Kyber, SABER, and NTRU. *IACR TCHES*, 2022(4):718–750, 2022. doi:10.46586/tches.v2022.i4.718-750.

[Hwa24] Vincent Hwang. A survey of polynomial multiplications for lattice-based cryptosystems. *IACR Commun. Cryptol.*, 1(2):1, 2024. URL: https://doi.org/10.62056/a0ivr-10k, doi:10.62056/A0IVR-10K.

[HYS+25] Syed Mahbub Hafiz, Bahattin Yildiz, Marcos A. Simplício Jr., Thales B. Paiva, Henrique S. Ogawa, Gabrielle De Micheli, and Eduardo Lopes Cominetti. Incompleteness in number-theoretic transforms: New tradeoffs and faster lattice-based cryptographic applications. In *2024 IEEE European Symposium on Security and Privacy*, pages 565–584. IEEE Computer Society Press, June / July 2025. doi:10.1109/EuroSP63326.2025.00039.

[HZZ+22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Improved plantard arithmetic for lattice-based cryptography. *IACR TCHES*, 2022(4):614–636, 2022. doi:10.46586/tches.v2022.i4.614-636.

[KPR+] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[Kre24] Katharina Kreuzer. Verification of correctness and security properties for CRYSTALS-KYBER. In *37th IEEE Computer Security Foundations Symposium, CSF 2024, Enschede, Netherlands, July 8-12, 2024*, pages 511–526. IEEE, 2024. doi:10.1109/CSF61375.2024.00016.

[KSSW22] Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. Improving software quality in cryptography standardization projects. In *IEEE European Symposium on Security and Privacy, EuroS&P 2022 - Workshops, Genoa, Italy, June 6-10, 2022*, pages 19–30, Los Alamitos, CA, USA, 2022. IEEE Computer Society. URL: https://eprint.iacr.org/2022/337, doi:10.1109/EuroSPW55150.2022.00010.

[Nat24a] National Institute of Standards and Technology. Module-lattice-based digital signature standard. Technical report, National Institute of Standards and Technology, 2024. URL: https://doi.org/10.6028/NIST.FIPS.204.

[Nat24b] National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard. Technical report, National Institute of Standards and Technology, 2024. URL: https://doi.org/10.6028/NIST.FIPS.203.

[Nat24c] National Institute of Standards and Technology. Stateless hash-based digital signature standard. Technical report, National Institute of Standards and Technology, 2024. URL: https://doi.org/10.6028/NIST.FIPS.205.

[PFH+20] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[Pla21] Thomas Plantard. Efficient word size modular arithmetic. *IEEE Trans. Emerg. Top. Comput.*, 9(3):1506–1518, 2021. doi:10.1109/TETC.2021.3073475.

[PTWY18] Andy Polyakov, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verifying arithmetic assembly programs in cryptographic primitives (invited talk). In Sven Schewe and Lijun Zhang, editors, *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*, volume 118 of *LIPIcs*, pages 4:1–4:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. URL: https://doi.org/10.4230/LIPIcs.CONCUR.2018.4, doi:10.4230/LIPICS.CONCUR.2018.4.

[SAB+20] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[Thé22] Laurent Théry. A formalisation of a fast fourier transform. *CoRR*, abs/2210.05225, 2022. URL: https://doi.org/10.48550/arXiv.2210.05225, arXiv:2210.05225, doi:10.48550/ARXIV.2210.05225.

[ZBPB17] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In

Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1789–1806. ACM Press, October / November 2017. doi:10.1145/3133956.3134043.