Efficient Embedded Security Standards (EESS)

EESS #1: Implementation Aspects of NTRUEncrypt and NTRUSign

Consortium for Efficient Embedded Security

June 20th, 2003

Version 2.0

Comments regarding this document are welcomed by the editor, William Whyte, wwhyte@ntru.com.

Table of Contents

1 Introduction

1.1 Scope

This document specifies common techniques and implementation choices using the NTRUEncrypt and NTRUSign public-key cryptography algorithms. Topics covered include:

- Cryptographic primitives – The building blocks for a secure cryptographic scheme
- Cryptographic schemes – Complete sequences of operations for performing secure cryptographic functions
- Supported parameter choices – Specific selections of approved sets of values for cryptographic parameters
- Certificate formats – Definition of fields for data structures that cryptographically bind a public key (and other information) to an entity
- ASN.1 syntax for NTRUEncrypt and NTRUSign – Standard formats of cryptographic data items

In addition, this standard includes relevant information to assist in the development and interoperable implementation of NTRUEncrypt and NTRUSign, including security considerations and test vectors.

1.2 Purpose

Enormous investments in wireless and consumer infrastructures mandate the need for stronger, more efficient security. First-generation security solutions offer inadequate efficiency and scalability to meet the requirements of mass-market adoption of wireless and embedded consumer applications. To address this need, new security infrastructures are emerging and must be carefully, but rapidly, defined.

In order to ensure interoperability within wired and wireless environments and allow for the rapid deployment of emerging security infrastructures, the Consortium for Efficient Embedded Security (CEES) began work on the Efficient Embedded Security Standards (EESS) in order to provide universal specifications for creating secure, interoperable implementations of highly efficient, highly scalable public-key security.

CEES intends that the EESS will combine the experience and knowledge of experts in academia as well as in commercial industry to provide a complete specification of well-studied, efficient and interoperable methodologies using modern public-key techniques. EESS #1 is designed to specify highly efficient public-key cryptographic techniques that can be used in highly scalable secure applications.

1.3 Compliance

Implementations may claim compliance with the cryptographic schemes included in this standard provided the external interface (input and output) to the schemes is identical to the interface specified in this document and the supported encoding methods and parameter selections are used. Internal computations may be performed as specified in this document, or may be performed via an equivalent sequence of operations. In this document, the word “shall” implies a requirement for an implementation to meet the standard, while the word “should” denotes a choice left to the implementer.

1.4 EESS Publication Guidelines

CEES maintains control over the contents and publication of the EESS series. In order to promote an open standards process, the documents will be made available to the public on the CEES web site at www.ceesstandards.org. In addition, the Consortium welcomes input from the community at large. Comments may be submitted to the editor, William Whyte, at wwhyte@ntru.com.

1.5 Intellectual Property

Compliance with this standard, any other CEES standard, or any standard referenced herein may be subject to intellectual property claims by third parties. By publication of this document, CEES takes no position with respect to the validity of such claims. When possible, the CEES has made efforts to obtain information relating to patent coverage of techniques included in EESS.

In particular, NTRU Cryptosystems, Inc. has been granted U.S. Patent No. 6,081,597, which covers aspects of the NTRUEncrypt public-key encryption scheme, and has applied for a patent (or patents) that covers the NTRUSign public-key signature scheme. In addition, NTRU Cryptosystems may have applied for additional patent coverage on implementation techniques defined in this standard.

2 Mathematical Foundations

The cryptographic techniques specified in this standard require arithmetic in quotient polynomial rings, also called convolution polynomial rings, defined in section 2.2.3. Intuitively, these algebraic objects consist of polynomials with integer coefficients. Manipulation of these ring elements is accomplished by polynomial arithmetic modulo a fixed polynomial: $X^N - 1$ in this standard. Careful selection of parameters allows fast implementation of these operations on a microprocessor. For typical values of N, operations in these rings can be performed with 8-bit shifts, adds and multiplies. The simplicity of the operations and the ability to perform coefficient operations in parallel account for the high speed, small footprint and ease of deployment on constrained devices.

This section includes mathematical background for the techniques in the standard and provides data conversion methods for use with the cryptographic algorithms.

2.1 Conventions and Notation

2.1.1 Notation

When referring to mathematical objects and data objects in this standard, the following notation is used. Note that throughout the document, numbers are used to distinguish different, but related values (e.g. *df1*, *df2*, *df3*).

0	Denotes the integer 0, the bit 0, or the additive identity (the element zero) of a ring
1	Denotes the integer 1, the bit 1, or the multiplicative identity (the element one) of a ring
*	Indicates the convolution product operation of two polynomials and is also used to indicate multiplication in the integers
X	The indeterminate used in polynomials
$\mathbf{Z}$	The ring of integers
mod q	Used to reduce the coefficients of a polynomial into some interval of length q
mod p	Used to reduce a polynomial to a representative of the polynomial ring modulo p
N	Dimension of the polynomial ring used (i.e. polynomials are up to degree N-1)
p	"Small" modulus, an integer or a polynomial
q	"Big" modulus, usually an integer
h	NTRUEncrypt or NTRUSign public key
r	Encryption blinding value (generated from the hash of the message m)
f	NTRUEncrypt private key; part of NTRUSign private key
g	Temporary polynomial used in the key generation process in NTRUEncrypt; optional part of NTRUSign private key.

F	In NTRUEncrypt, a polynomial that is used (and is often sufficient) to calculate the value f; in NTRUSign, part of the basis completion space, stored with the private key.
G	In NTRUEncrypt, a polynomial that is used (and is often sufficient) to calculate the value g; in NTRUSign, part of the basis completion space, optionally stored with the private key.
i	Message representative, a polynomial, computed by a message encoding operation
e	Encrypted message, a polynomial, computed by an encryption primitive
m	The message, an octet string, which is encrypted in an encryption scheme or whose signature is computed by a signature scheme
$f*h$	The convolution product of f and h, where f and h are polynomials
dr	An integer specifying the number of ones in the blinding value r
df	An integer specifying the number of ones in the polynomials that comprise the private key value f (usually specified as df_1, df_2, and df_3, or as dF)
dg	An integer specifying the number of ones in the polynomials that comprise the temporary polynomial g (often specified as dG)
db	The number of random bits used as input for encryption
D_f	The space of allowable values of the polynomial f (there are also spaces for g, r, h, etc. denoted by D_g, D_r, D_h, etc.)
A	NTRUEncrypt average decryption coefficient, used in decryption process to reduce into correct interval
T	Wrapping tolerance value used to determine when the decryption process fails
ceil[]	Ceiling function (i.e. the smallest integer greater than or equal to the contents of [])
floor[]	Floor function (i.e. the largest integer less than or equal to the contents of [])
Hash( )	A cryptographic hash function computed on the contents of ( )
PRNG( )	A pseudo-random number generation function seeded with the contents of ( )
MGF( )	A mask generation function seeded with the contents of ( )
$A\|\|B$	Concatenation of the octet strings A and B where the leading octet of A is the leading octet of $A\|\|B$ and the trailing octet of B is the trailing octet of $A\|\|B$.
$a := b$	Initialize or set the value of a equal to the value of b.
DesEncrypt(o,K)	The result of encrypting the octet string o with the DES algorithm [FIP99] under key K.

2.1.2 Bit Strings and Octet Strings

As usual, a **bit** is defined to be an element of the set {0, 1}. A **bit string** is defined to be an ordered array of bits. A **byte** (also called an **octet**) is defined to be a bit string of length 8. A **byte string** (also called an **octet string**) is an ordered array of bytes. The terms **first** and **last**, **leftmost** and **rightmost**, **most significant** and **least significant**, and

leading and **trailing** are used to distinguish the ends of these sequences (**first**, **leftmost**, **most significant** and **leading** are equivalent; **last**, **rightmost**, **least significant** and **trailing** are equivalent). Within a byte, we additionally refer to the **high-order** and **low-order** bits, where **high-order** is equivalent to **first** and **low-order** is equivalent to **last**.

Note that when a string is represented as a sequence, it may be indexed from left to right or from right to left, starting with any index. For example, consider the octet string of two octets: 2a 1b. This corresponds to the bit string 0010 1010 0001 1011. No matter what indexing system is used, the first octet is still 2a, the first bit is still 0, the last octet is still 1b, and the last bit is still 1. The high-order bit of the second octet is 0; the low-order bit of the second octet is 1.

In this standard, a bit string or an octet string may be used to represent a polynomial with coefficients reduced mod q, where q is usually either 128 or 256. In this case, the integer coefficients are mapped individually to bit or octet strings, which are then concatenated. Allowable mappings and their reverses are described in the conversion primitives OS2REP, BS2REP, POS2REP, RE2OSP, RE2BSP and RE2POSP in sections 2.3.4, 2.3.5 and 2.3.6.

When a bit string or an octet string is used to represent a polynomial with binary coefficients, for reasons of efficiency we use a mapping that is different from simply left-to-right or right-to-left translation of bits into polynomial coefficients. This mapping and its reverse are described for octet strings in the conversion primitives OS2BEP and BE2OSP in section 2.3.7. This standard does not specify a means of converting between bit strings and binary ring elements.

2.1.3 Algorithm Specification Conventions

When specifying an algorithm or method, this standard uses four parts to specify different aspects of the algorithm. They are as follows:

- Components, such as choice of PRNG, are parameters that are specified before the beginning of the operation and that are not specific to the particular algorithm call. Components tend to be kept fixed for multiple users and multiple instances of the algorithm call and need not be explicitly specified if they are implicitly known (e.g. if they are defined within a selected object identifier (OID)).
- Inputs, such as keys and messages, are values that must be specified for each algorithm call.
- Outputs, such as ciphertext, are the result of transformations on the inputs.
- Operations specify the transformations that are performed on the data to arrive at the output. Throughout the standard, the operations are defined as a sequence of steps. A conformant implementation may perform the operations using any sequence of steps that always produces the same output as the sequence in this standard. Caution should be taken to ensure that intermediate values are not revealed, however, as they may compromise the security of the algorithms.

2.2 Convolution Polynomial Ring Representation and Arithmetic

This section describes the representation and arithmetic of quotient ring elements used throughout the EESS #1 standard.

2.2.1 Modular Operations on Integers

This standard uses integer modular arithmetic in several instances, including operations on the coefficients in the polynomial representation of a quotient ring element. These modular operations are performed in the canonical manner, taking the remainder when the intermediate result is divided by the modulus. In this standard, the representatives of the congruence classes modulo q are frequently taken to be different from the usual set of equivalence representatives $[0, 1, 2, \ldots, q-1]$. When appropriate, the range of representatives for modular reduction is specified explicitly.

2.2.2 Representation of Polynomials

Typically in mathematical literature, a polynomial a in X is denoted $a(X)$. In this standard, when the meaning is clear from the context, polynomials a in the variable X will simply be denoted a. Further, all polynomials used in this standard have degree $N-1$, unless otherwise noted. In addition, given a polynomial a, a variable denoted a_i, where i is an integer, represents the coefficient of a of degree i. In other words, the polynomial denoted a represents the polynomial $a(X) = a_0 + a_1X^1 + a_2X^2 + a_3X^3 + \ldots + a_{N-1} X^{N-1}$, unless otherwise specified.

2.2.3 Convolution Polynomial Rings Over the Integers

Let $\mathbf{Z}$ be the ring of integers. The polynomial ring over $\mathbf{Z}$, denoted $\mathbf{Z}[X]$, is the set of all polynomials with coefficients in the integers. The *convolution polynomial ring (over* $\mathbf{Z}$*) of degree* N is the quotient ring $\mathbf{Z}[X]/(X^N - 1)$. The product c of two polynomials a,b ε $\mathbf{Z}[X]/(X^N - 1)$ is given by the formula

$$c(X) = a(X) * b(X) \quad \text{with} \quad c_k = \sum_{i+j \equiv k \pmod N} a_i b_j \ .$$

Identifying polynomials $a = a_0 + a_1X + a_2X^2 + \ldots + a_{N-1} X^{N-1}$ with their coefficient vectors $[a_0, a_1, a_2, \ldots, a_{N-1}]$, this convolution product formula makes the space of N-tuples $\mathbf{Z}^N$ into a ring. The convolution polynomial ring may thus be identified with the ring of N-tuples with convolution product.

Note that in EESS #1, all multiplications of polynomials a and b, represented as $a*b$, are taken to occur in the ring $\mathbf{Z}[X]/(X^N - 1)$ unless otherwise noted.

The convolution polynomial rings $(\mathbf{Z}/p\mathbf{Z})[X]/(X^N - 1)$ and $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$ are examples of the above, with arithmetic operations performed as specified in the following sections.

2.2.4 Basic Convolution Polynomial Ring Arithmetic

Since ring elements are represented by polynomials, canonical techniques for polynomial arithmetic may be used with added steps for coefficient and polynomial modular reduction. These steps are specified in sections 2.2.5 and 2.2.6, respectively. The rule for multiplication is given in section 2.2.3.

2.2.5 Reduction of a Polynomial mod *q*

Throughout the document, polynomials are taken *mod q*, where q is an integer. To reduce a polynomial *mod q*, one simply reduces each of the coefficients independently *mod q* into the appropriate (specified) interval.

2.2.6 Inversion in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$

For certain cryptographic operations such as key generation, it is necessary to take the inverse of a polynomial in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$. Section 2.2.8.4 gives an algorithm for doing this using the algorithms defined in sections 2.2.8.1 - 2.2.8.3. This algorithm works for q a power of a prime (q is often a power of 2).

2.2.6.1 The Polynomial Division Algorithm in $\mathbf{Z}_p[X]$

This algorithm divides one polynomial by another polynomial in the ring of polynomials with integer coefficients modulo a prime p. All convolution operations occur in the ring $\mathbf{Z}_p[X]$ in this algorithm (i.e. there is no modular reduction of the powers of the polynomials).

Input: A prime p, a polynomial a in $\mathbf{Z}_p[X]$ and a polynomial b in $\mathbf{Z}_p[X]$ of degree N whose leading coefficient b_N is not 0.

Output: Polynomials q and r in $\mathbf{Z}_p[X]$ satisfying $a = b * q + r$ and deg $r <$ deg b.

```
1. Set $r := a$ and $q := 0$
2. Set $u := b_N^{-1} \bmod p$
3. While deg $r >= N$ do
   3.1. Set $d :=$ deg $r(X)$
   3.2. Set $v := u*r_d*X^{(d-N)}$
   3.3. Set $r := r - v * b$
   3.4. Set $q := q + v$
4. Return $q, r$
```

2.2.6.2 The Extended Euclidean Algorithm in $\mathbf{Z}_p[X]$

The Extended Euclidean Algorithm finds a greatest common divisor d (there may be more than one that are constant multiples of each other) of two polynomials a and b in $\mathbf{Z}_p[X]$ and polynomials u and v such that $a*u + b*v = d$. All convolution operations occur in the ring $\mathbf{Z}_p[X]$ in this algorithm (i.e. there is no modular reduction of the powers of the polynomials).

Input: A prime p and polynomials a and b in $\mathbf{Z}_p[X]$ with a and b not both zero

Output: Polynomials u, v, d in $\mathbf{Z}_p[X]$ with $d = \mathrm{GCD}(a, b)$ and $a*u + b*v = d$

```
1. If $b = 0$ then return $(1,0,a)$
2. Set $u := 1$
3. Set $d := a$
4. Set $v_1 := 0$
5. Set $v_3 := b$
```

```
6. While $v_3 \neq 0$ do
   6.1. Use the division algorithm (section 2.2.8.1) to write $d = v_3*q + t_3$ with deg $t_3 <$ deg $v_3$
   6.2. Set $t_1 := u - q*v_1$
   6.3. Set $u := v_1$
   6.4. Set $d := v_3$
   6.5. Set $v_1 := t_1$
   6.6. Set $v_3 := t_3$
7. Set $v := (d - a*u)/b$  [This division is exact, i.e., the remainder is 0]
8. Return $(u,v,d)$
```

2.2.6.3 Inverses in $\mathbf{Z}_p[X]/(X^N - 1)$

The Extended Euclidean Algorithm may be used to find the inverse of a polynomial a in $\mathbf{Z}_p[X]/(X^N - 1)$ if the inverse exists. The condition for the inverse to exist is that GCD(a, $X^N - 1$) should be a polynomial of degree 0 (i.e. a constant). All convolution operations occur in the ring $\mathbf{Z}_p[X]/(X^N - 1)$ in this algorithm.

Input: A prime p, a positive integer N and a polynomial a in $\mathbf{Z}_p[X]/(X^N - 1)$

Output: A polynomial b satisfying $a*b = 1$ in $\mathbf{Z}_p[X]/(X^N - 1)$ if a is invertible in $\mathbf{Z}_p[X]/(X^N - 1)$, otherwise FALSE

```
1. Run the Extended Euclidean Algorithm (Section 2.2.8.2) with input $a$ and $(X^N - 1)$. Let $(u, v, d)$ be the
   output, such that $a*u + (X^N - 1)*v = d = \mathrm{GCD}(a, (X^N - 1))$.
2. If deg $d = 0$
   2.1. Return $b = d^{-1} \pmod{p} * u$
3. Else return FALSE
```

2.2.6.4 Inverses in $\mathbf{Z}_q[X]/(X^N - 1)$

This algorithm finds the inverse of a polynomial a in $\mathbf{Z}_q[X]/(X^N - 1)$, where q is a power of a prime. In particular, this is used to calculate the inverse of a convolution polynomial mod q for NTRUEncrypt key generation where q is a power of 2.

Input. A prime p, an exponent e such that $p^e = q$, a positive integer N and a polynomial a in $\mathbf{Z}_q[X]/(X^N - 1)$.

Output. An inverse b of a in the ring $\mathbf{Z}_q[X]/(X^N - 1)$ if the inverse exists, otherwise FALSE.

```
1. Use the Inversion Algorithm (Section 2.2.8.3) to compute a polynomial $b$ that gives an inverse of $a(X)$
   in $\mathbf{Z}_p[X]/(X^N - 1)$. Return FALSE if the inverse does not exist.
2. Set $n := 2$
3. While $e > 0$ do
   3.1. Set $b := 2*b - a*b^2$  in $\mathbf{Z}[X]/(X^N - 1)$, with coefficients computed modulo $p^n$
   3.2. Set e :=floor[$e/2$]
   3.3. Set $n := 2*n$
4. Return $b(X)$ in $\mathbf{Z}[X]/(X^N - 1)$ with coefficients computed modulo $p^e = q$.
```

2.2.7 Resultant Generation

In order to calculate a complete NTRUSign private key, it may be necessary to compute the resultant of a polynomial with the NTRU polynomial $X^N - 1$. Computing resultants over large polynomials may require a large amount of memory and computation. In order to improve the efficiency of computing the resultant, it may be desirable to compute the resultant modulo smaller primes and lift the results into the integers.

2.2.7.1 Resultant of a polynomial with $X^N - 1$ mod p

The resultant of a polynomial P modulo a reasonably small prime p may be calculated during key generation in order to find the resultant without the intermediate coefficients growing too big. Note that this algorithm finds a single polynomial *rhoP* of degree less than N such that P**rhoP* mod p is equal to the resultant mod $X^N - 1$. The multiple of $X^N - 1$ (denoted *rhox*) can be computed easily from P and *rhoP* if desired, but is usually not needed and hence is omitted from the algorithm below. Note that the degree of *rhoP* will not be greater than $N - 1$, even though the computations are performed in $\mathbf{Z}_p[X]$.

Input: A prime p, an integer N and a polynomial P in $(\mathbf{Z}/p\mathbf{Z})[X]/(X^N - 1)$

Output: Polynomial *rhoP* in $(\mathbf{Z}/p\mathbf{Z})[X]/(X^N - 1)$ and an integer *resultant* satisfying $resultant = rhoP * P + rhox *(X^N - 1)$ in $\mathbf{Z}_p[X]$ for some *rhox* in $\mathbf{Z}_p[X]$.

```
1. Set polynomials $A := X^N - 1$, $B := P$
2. Set polynomials $V1 := 0$, $V2 := 1$, $Temp := 0$
3. Set integers $a := \deg A$, $b := \deg B$, $tempa := \deg A$, $c := 0$, $resultant := 1$
4. While $b > 0$ do
    a. Set $c := B_b^{-1} * A_a \bmod p$
    b. Set $A := A - c*B*X^{(a-b)}$ in $\mathbf{Z}_p[X]$
    c. Set $V1 := V1 - V2*c*X^{(a-b)}$ in $\mathbf{Z}_p[X]$
    d. If $\deg A < b$
        i. Set $resultant := resultant*B_b^{(\mathrm{temp}a-(\deg A))} \bmod p$
        ii. If $tempa$ and $b$ are both odd
            1. Set $resultant := -resultant \bmod p$
        iii. Set $Temp := A$
        iv. Set $A := B$
        v. Set $B := Temp$
        vi. Set $Temp := V1$
        vii. Set $V1 := V2$
        viii. Set $V2 := Temp$
        ix. Set $tempa := b$
    e. Set $a := \deg A$, $b := \deg B$
5. Set $resultant := resultant*B_0^a \bmod p$
6. Set $c := B_0^{-1} \bmod p$
7. Set $rhoP := V2*c*resultant \bmod p$
8. Return $rhoP$, $resultant$
```

2.2.7.2 Resultant of a polynomial with $X^N - 1$

The resultant of a polynomial P with $X^N - 1$ may be computed by computing the resultant modulo a list of reasonably small primes and combining the results to obtain the resultant over the integers. If the product of the primes is greater than the maximum possible resultant value, the resultant will always be obtained exactly. Note that this algorithm finds a single polynomial *rhoP* of degree less than N such that P**rhoP* is equal to the resultant mod $X^N - 1$. The multiple of $X^N - 1$ (denoted *rhox*) can be computed easily from P and *rhoP* if desired, but is usually not needed and hence is omitted from the algorithm below.

Input: A prime p, an integer N and a polynomial P in $\mathbf{Z}[X]/(X^N - 1)$

Output: Polynomial *rhoP* in $\mathbf{Z}[X]/(X^N - 1)$ and an integer *resultant* satisfying $resultant = rhoP * P + rhox *(X^N - 1)$ in $\mathbf{Z}_p[X]$ for some *rhox* in $\mathbf{Z}_p[X]$.

```
1. Select an integer $Max$ greater than the largest possible value of the resultant (this value may be calculated ahead of time and simply retrieved)
2. Select a set of $m$ distinct primes $p1, p2, p3, \ldots, pm$ such that $p1*p2*p3*\ldots*pm > 2*Max$
3. Set integer $pproduct := 1$, $resultant := 1$
4. Set polynomial $rhoP := 1$
5. Set integer $j := 0$, $temp := 0$
6. While $j < m$ do
    a. Set $temp := pj*pprod$ (e.g. for $j = 0$, set polynomial $temp$ equal to $p0*pprod$, for $j = 1$, set polynomial $temp$ equal to $p1*pprod$, etc.)
    b. Calculate integer $resp$ and polynomial $rhop$ such that $rhop*P = resp$ in $(\mathbf{Z}/pj\mathbf{Z})[X]/(X^N - 1)$ (using algorithm 2.2.7.1)
    c. Find integers $alphap$ and $betapprod$ such that $alphap*pj + betapprod*pprod = 1$ (using Extended Euclidean Algorithm in the integers)
    d. Set $resultant := resultant*alphap*pj + resp*betapprod*pprod \pmod{temp}$
    e. Set $rhoP := rhoP*alphap*pj + rhop*betapprod*pprod$ in $(\mathbf{Z}/temp\mathbf{Z})[X]/(X^N - 1)$
    f. Set $pprod := temp$
    g. Set $j := j + 1$
7. Set $rhoP := rhoP \pmod{2*Max}$ with coefficients reduced into the range $(–Max, Max]$
8. Set $resultant := rhoP \pmod{2*Max}$ reduced into the range $(–Max, Max]$
Output $rhoP$ and $resultant$
```

2.3 Data Types and Conversions

This section describes the primitives that shall be used to convert between different types of objects and strings when such conversion is required in primitives, schemes or encoding techniques.

2.3.1 Converting Between Bit Strings and Octet Strings (BS2OSP and OS2BSP)

To represent a bit string as an octet string, one simply appends enough zeroes following the last bit to make the number of bits a multiple of 8, and then breaks it up into octets. More precisely, a bit string $b_0\, b_1 \ldots b_{l-1}$ of length l shall be converted to an octet string $M_0\, M_1 \ldots M_{d-1}$ of length $d = \lceil l/8 \rceil$ as follows: for $0 \le i < d - 1$, let the octet $M_i = b_{8i}\, b_{8i+1} \ldots b_{8i+7}$. The final octet M_{d-1} shall have its low-order $8d - l$ bits set to 0; its high-order $8 - (8d - l)$ bits shall be $b_{8d-8}.\ b_{8d-7} \ldots b_{l-1}$.

The primitive that converts bit strings to octet strings is called Bit String to Octet String Conversion Primitive or BS2OSP. It takes the bit string as input and outputs the octet string.

The primitive that converts octet strings to bit strings is called Octet String to Bit String Conversion Primitive or OS2BSP. It takes an octet string of length d and the desired length l of the bit string as input. It shall output the bit string if $d = \lceil l/8 \rceil$ and if the final $8d - l$ bits of the final octet are zero; it shall output “error” otherwise.

2.3.2 Converting Between Integers and Octet Strings (I2OSP and OS2IP)

To represent a non-negative integer x as an octet string of length l (l has to be such that $256^l > x$), the integer shall be written in its unique l-digit representation base 256:

$$x = x_{l-1}\, 256^{l-1} + x_{l-2}\, 256^{l-2} + \ldots + x_1\, 256 + x_0$$

where $0 \le x_i < 256$ (note that one or more leading digits will be zero if $x < 256^{l-1}$). Then let the octet M_i have the value x_i for $0 \le i \le l\text{-}1$. The octet string shall be $M_{l\text{-}1}\ M_{l\text{-}2}\ \ldots\ M_0$.

For example, the integer 10945 is represented by an octet string of length 3 as 00 2A C1.

The primitive that converts integers to octet strings is called Integer to Octet String Conversion Primitive or I2OSP. It takes an integer x and the desired length l as input and outputs the octet string if $256^{\,l} > x$. It shall output "error" otherwise.

The primitive that converts octet strings to integers is called Octet String to Integer Conversion Primitive or OS2IP. It takes an octet string as input and outputs the corresponding integer. Note that the octet string of length zero (the empty octet string) is converted to the integer 0.

2.3.3 Converting Between Integers and Bit Strings (I2BSP and BS2IP)

To represent a non-negative integer x as a bit string of length l (l has to be such that $2^l > x$), the integer shall be written in its unique l-bit binary representation:

$$x = x_{l-1}\, 2^{l-1} + x_{l-2}\, 2^{l-2} + \ldots + x_1\, 2 + x_0$$

where x_i is 0 or 1 (note that one or more leading bits will be zero if $x < 256^{\,l-1}$). Then let the bit b_i have the value x_i for $0 \le i \le l\text{-}1$. The bit string shall be $b_{l\text{-}1}\ b_{l\text{-}2}\ \ldots\ b_0$.

For example, the integer 10945 is represented by a bit string of length 20 as 0000 0010 1010 1100 0001.

The primitive that converts integers to bit strings is called Integer to Bit String Conversion Primitive or I2BSP. It takes an integer x and the desired length l as input and outputs the bit string if $2^l > x$. It shall output "error" otherwise.

The primitive that converts bit strings to integers is called Bit String to Integer Conversion Primitive or BS2IP. It takes a bit string as input and outputs the corresponding integer. Note that the bit string of length zero (the empty bit string) is converted to the integer 0.

2.3.4 Converting Between Ring Elements and Octet Strings (RE2OSP and OS2REP)

An element a of a convolution polynomial ring, for the purposes of this standard, is represented by an array of N integers. In this standard, the "big" modulus q is always less than or equal to 256, so each of the N coefficients of a polynomial that is taken mod q may be represented as a single octet. To represent a as an octet string, I2OSP is used to produce a one-octet encoding of the integer value a_i of each coefficient of a in turn. The coefficients are encoded in increasing order starting with the constant coefficient and ending with the coefficient of $X^{N\text{-}1}$. The results of this conversion are placed from least significant to most significant in an octet string of length N. For example, if q=128 and N=5, the polynomial

$$a[X] = 45 + 2X + 77\, X^2 + 103\, X^3 + 12\, X^4$$

is represented by the octet string 2d 02 4d 67 0c.

The primitive that converts ring elements to octet strings is called Ring Element to Octet String Conversion Primitive or RE2OSP. It takes a ring element *a* and the degree *N* as inputs and outputs the corresponding octet string.

To convert an octet string back to a field element, if *q* is less than or equal to 256, then OS2IP shall be used on each octet going from least significant to most significant and the result is taken to be a coefficient of the polynomial *a* in increasing order starting with the coefficient of lowest degree.

The primitive that converts octet strings to ring elements is called Octet String to Ring Element Conversion Primitive or OS2REP. It takes the octet string, the “big” modulus *q* as inputs and outputs the corresponding ring element. It shall output “error” if OS2IP outputs “error.”

2.3.5 Converting Between Ring Elements and Bit Strings (RE2BSP and BS2REP)

While octet string representation may be most convenient for ring element arithmetic in a microprocessor, ring elements may be more compactly stored and transmitted as bit strings. To represent a ring element *a* as a bit string, the modulus *q* is required. I2BSP is used on each coefficient of *a* in turn to produce a ceil[$log_2\, q$]-bit encoding of the integer value a_i. The coefficients are encoded in increasing order starting with the constant coefficient and ending with the coefficient of X^{N-1}. The results of this conversion are placed from least significant to most significant in an bit string of length *N* ceil[$log_2\, q$]. For example, if q=128 and N=5, the polynomial

$$a[X] = 45 + 2X + 77\, X^2 + 103\, X^3 + 12\, X^4$$

is represented by the bit string 0101101 0000010 1001101 1100111 0001010. (If this were subsequently to be converted to an octet string using BS2OSP, it would become first the bit string 0101 1010 0000 1010 0110 1110 0111 0001 0100 0000, and then the octet string 5a 0a 6e 71 40).

The primitive that converts ring elements to bit strings is called Ring Element to Bit String Conversion Primitive or RE2BSP. It takes a ring element *a*, the degree *N*, and the big modulus *q* as inputs and outputs the corresponding bit string.

To convert a bit string to a ring element, the modulus *q* is required. Convert each group of ceil[$log_2\, q$] bits to an integer using BS2IP, starting with the least significant bits and going to the most significant bits, and set each coefficient from lowest degree to highest degree to be the integer produced.

The primitive that converts bit strings to ring elements is called Bit String to Ring Element Conversion Primitive or RE2BSP. It takes a bit string and the modulus q as inputs and outputs the corresponding ring element.

2.3.6 Converting Between Ring Elements and Packed Octet Strings (RE2POSP and POS2REP)

To save space when converting between ring elements and octet strings, the ring element may instead be converted to a packed octet string. This conversion, and its inverse, are performed using the following primitive.

2.3.6.1 Ring Element to Packed Octet String Conversion Primitive (RE2POSP)

Input:
— A ring element e
— The ring parameters q, N.

Output:
— The octet string representation o of e.

Operation:
1. Convert e to a bit string b, using RE2BSP with inputs e, N, and q.
2. Convert b to an octet string o using BS2OSP with input b.
3. Output o.

2.3.6.2 Packed Octet String to Ring Element Conversion Primitive (POS2REP)

Input:
— The octet string representation o of e.
— The ring parameters q, N.

Output:
— A ring element e, or “error”.

Operation:
1. Convert i to a bit string b, using OS2BSP with inputs i, and $N * \lceil \log_2(q) \rceil$. If OS2BSP outputs “error”, output “error”.
2. Convert b to a ring element e using BS2REP with input b and q.
3. Output e.

2.3.7 Converting Between Binary Ring Elements and Octet Strings (ME2BSP and BS2MEP)

If an element in the ring $\mathbf{Z}[X]/(X^N - 1)$ is known to have all its coefficients to be 0 or 1, it can be encoded as an octet string more efficiently than by the method given above. One considers each octet as a bit string of length 8. Then for each bit in the octet, starting with the low-order bit and working to the high-order bit, one sets a coefficient of the ring element equal to 1 if the bit is a 1 and 0 if it is a 0, starting with the low-order bit in the first octet and the lowest degree coefficient (degree 0). The encoding fails if the length of the octet string is greater than ceil[N/8]. If N is not equal to 0 mod 8, the encoding also fails if any of the high-order ($8 - N$ mod 8) bits in the final octet are set.

More precisely, an octet string $o_0\ o_1 \dots o_{l\text{-}1}$ of length $l \le$ floor[N/8] shall be converted to a binary ring element $a = a_0 + a_1X + \dots + a_{N\text{-}1}X^{N\text{-}1}$ as follows. Consider each octet to be a bit string, indexed from low-order to high-order bit: thus $o_0\ o_1 \dots o_{l\text{-}1}$ becomes ($b_{0,7}\ b_{0,6} \dots b_{0,0}$) ($b_{1,7}\ b_{1,6} \dots b_{l1,0}$) … ($b_{l\text{-}1,7}\ b_{l\text{-}1,6} \dots b_{l\text{-}1,0}$). Then set $a_0 = b_{0,0}$, $a_1 = b_{0,1}$, …, $a_{8i+j} = b_{i,j}$, … , $a_{8l-1} = b_{l-1,7}$, and $a_{8l} = a_{8l+1} = \dots = a_{N\text{-}1} = 0$.

An octet string $o_0\ o_1 \dots o_{l\text{-}1}$ of length $l =$ ceil[N/8] shall be converted to a binary ring element as follows. Consider each octet to be a bit string, indexed from low-order to high-order bit: thus $o_0\ o_1 \dots o_{l\text{-}1}$ becomes ($b_{0,7}\ b_{0,6} \dots b_{0,0}$) ($b_{1,7}\ b_{1,6} \dots b_{l1,0}$) … ($b_{l\text{-}1,7}\ b_{l\text{-}1,6} \dots b_{l\text{-}1,0}$). If there exists an i such that $8l\text{-}8\text{+}i \ge N$ and $b_{l\text{-}1,i}$ is 1, output “error”. Otherwise, set $a_0 = b_{0,0}$, $a_1 = b_{0,1}$, …, $a_{8i+j} = b_{i,j}$, … , $a_{N-1} = b_{l-1,N \bmod 8}$.

An octet string $o_0\ o_1 \dots o_{l\text{-}1}$ of length $l >$ ceil[N/8] shall be converted to a binary ring element as follows. If there is any $i >$ ceil[N/8] such that the octet o_i is non-zero, output “error”. Otherwise, truncate the octet string to length ceil[N/8] by discarding the final octets and use the method given in the previous paragraph.

The primitive that converts octet strings to binary elements in the ring $\mathbf{Z}[X]/(X^N - 1)$ is called Octet String to Binary Element Conversion Primitive or OS2BEP. It takes the octet string as input and outputs the ring element. If the octet string cannot be encoded (because bits are set which would correspond to coefficients of powers of X greater than or equal to X^N) it shall output “error”.

The primitive that converts binary elements in the ring $\mathbf{Z}[X]/(X^N - 1)$ to octet strings is called Binary Element to Octet String Conversion Primitive or BE2OSP. It takes a binary element a in the ring $\mathbf{Z}[X]/(X^N - 1)$ and the desired length l of the octet string as input. It shall output the octet string if l is greater than or equal to ceil[N/8] or if all the coefficients of a of degree $8l$ or greater are 0; it shall output “error” otherwise.

3 Cryptographic Building Blocks

This section defines the building blocks for implementing the NTRUEncrypt and NTRUSign algorithms.

These building blocks have the following taxonomy:

- Components consist of domain parameters, security parameters and scheme options. Selections for all of the components must be made in order to properly implement the cryptographic schemes (defined in section 4). The NTRUEncrypt components are defined in section 3.1.
- Primitives are sequences of mathematical operations that are performed on inputs in order to perform a basic cryptographic function. Primitives are designed to require certain properties of inputs and ensure certain properties of outputs. They are not designed to provide overall security properties by themselves, but to provide security when used appropriately in the cryptographic schemes. Primitives are scheme option choices and are specified for NTRUEncrypt in section 3.2.
- Encoding methods are specific scheme option choices for how to transform data within the cryptographic scheme, but that are not intrinsically tied to the cryptographic primitives. It is possible that different encoding methods may be used in conjunction with the same cryptographic primitives, although in many cases they are closely related. Encoding methods are scheme option choices and are specified for NTRUEncrypt in section 3.3.
- Supporting algorithms are typically algorithms that are standardized by other standards bodies and that provide certain cryptographic properties that are desirable to provide security to the public-key schemes. The supporting algorithms may be used in the encoding methods, the primitives or directly in the schemes to provide security.

3.1 NTRUEncrypt Components

This section defines the NTRUEncrypt components, categorizes them (as domain parameters, security parameters or scheme options) and gives a basic description of the component. Security considerations for specific choices for the component are included when appropriate, however for detailed security considerations, see [IEEE P1363.1].

Instantiations of primitives and encoding methods (which are both scheme options) are specified in section 3.2 and section 3.3 respectively. Required choices for all NTRUEncrypt components are listed in section 4.3.

3.1.1 NTRUEncrypt Domain Parameters

Values for each of the domain parameters must be selected in order to define the space in which operations are performed in NTRUEncrypt. The domain parameters specified in this standard maximize efficiency and security. Note that some domain parameters or other choices may not need to be known to perform certain operations (e.g. in order to encrypt, one need not know the small modulus p explicitly).

3.1.1.1 NTRUEncrypt Degree

The degree N identifies the dimension of the convolution polynomial ring used. Although N is referred to here as the NTRUEncrypt degree, elements of the ring are represented as polynomials of degree $N – 1$.

The specific value of N is defined for each parameter set listed in section 4.3.

3.1.1.2 NTRUEncrypt Small Modulus

The small modulus p is used for key generation and for coefficient modular reduction (as described in section 2.2.6). The modulus p is also used implicitly in the blinding value generation methods and the message representative generation methods.

The specific value of p is defined for each parameter set listed in section 4.3.

3.1.1.3 NTRUEncrypt Big Modulus

As described in section 2.2.5, the big modulus q is used to define the larger polynomial ring used for NTRUEncrypt. The modulus q can generally be taken to be any value that is relatively prime in the ring to the small modulus p. Taking q to be equal to or slightly less than a power of 2 can result in faster modular arithmetic operations.

The specific value of q is defined for each parameter set listed in section 4.3.

3.1.2 NTRUEncrypt Security Parameters

Values for each of the security parameters may be globally specified or chosen by the holder of the private key. These values must be chosen from those specified in this standard and need not be kept secret. The security parameters specified in this standard are selected to maximize efficiency and security.

3.1.2.1 NTRUEncrypt Private Key Space

Generally, private keys may be chosen to be any small polynomial f such that f is invertible in both $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N – 1)$ and $(\mathbf{Z}/p\mathbf{Z})[X]/(X^N – 1)$. However, for improved efficiency, private keys in this standard shall be chosen of the form $f = 1 + pF$, where p is the small modulus and F is a polynomial in the space D_F. This restriction on the form of f removes the need to calculate the inverse of f mod p (since the inverse will always be 1), and hence key generation and decryption are faster. For security purposes, it is strongly recommended that D_F be large enough and the key be generated in a random enough manner to prevent brute force attacks. In the key generation method given in section 3.2.1.1, D_F is the space of all polynomials of degree $N – 1$ that have dF coefficients equal to 1 and the rest of the coefficients equal to 0.

Recommended values of dF are included in each parameter set listed in section 4.3. These parameters are selected to provide maximum security and efficiency. Note that, as mentioned above, f in the given form is always invertible in $(\mathbf{Z}/p\mathbf{Z})[X]/(X^N - 1)$ and that, with $f(1)$ relatively prime to q, f will almost always be invertible in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$.

3.1.2.2 NTRUEncrypt Temporary Polynomial Space

In the generation of an NTRUEncrypt key, the private key f and a temporary polynomial g are needed to generate the public key. The temporary polynomial g is chosen from the temporary polynomial space D_g. The polynomial g shall be chosen randomly by the entity performing key generation from the temporary polynomial space D_g. D_g is the space of all polynomials of degree $N - 1$ with dg coefficients equal to 1 and the rest of the coefficients equal to 0.

Recommended values of dg are included in each parameter set listed in section 4.3. These parameters are selected to provide maximum security and efficiency.

3.1.2.3 NTRUEncrypt Public Key Space

The NTRUEncrypt public key space is uniquely determined by the NTRUEncrypt private key space and NTRUEncrypt temporary polynomial space and hence need not be specified explicitly as a security parameter. The NTRUEncrypt public key space D_h consists of all polynomials h of degree $N - 1$ with $h = f^{-1} * g * p$ with coefficients reduced modulo q, where f, g are chosen from D_f and D_g respectively, p is the small modulus as defined above, and f^{-1} is the polynomial with coefficients reduced mod q such that $f^{-1} * f = f * f^{-1} = 1$ in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$. Note that it appears to be a very difficult problem to determine whether a given polynomial h is in the public-key space or not.

3.1.3 NTRUEncrypt Scheme Options

NTRUEncrypt scheme options consist of parameters and algorithms that do not affect the key space (i.e. that are not domain parameters), but that must be agreed upon in order to implement the NTRUEncrypt encryption scheme. Scheme options include the chosen primitives and encoding methods and the parameters that are needed to completely specify the encoding methods and primitives.

3.1.3.1 NTRUEncrypt Random Component Size

The NTRUEncrypt random component size db is the number of random bits that shall be used as input to the message representative generation method and its inverse. This value is chosen to protect the ciphertext from dictionary attacks and to make the encryption process non-deterministic.

The specific value of db is defined for each parameter set listed in section 4.3.

3.1.3.2 NTRUEncrypt Blinding Value Space

The NTRUEncrypt blinding value r is determined by the message and is chosen deterministically from the NTRUEncrypt blinding value space D_r. The blinding value is an input to both the encryption primitive and the decryption primitive. This standard specifies one method for choosing r, which takes D_r to be the space of all polynomials of degree N with dr coefficients equal to 1 and the rest of the coefficients equal to 0. See section 3.3.1.1 for blinding value generation methods.

Specific permitted values of *dr* are defined for each parameter set listed in section 4.3.

3.1.3.3 NTRUEncrypt Random Polynomial Generation Constant

We generate a binary polynomial with *d* 1s by generating *d* distinct indices mod *N*. However, the pseudo-random number generators defined in this standard generate octet strings. In order to prevent any bias in converting these random octets to random mod *N* integers, values that would cause bias are thrown out. For example, with *N* =251, each random octet is taken to be an integer mod 256, and values of *N* or higher are discarded. (This is a bias because there are two ways of generating a value of “1” – 1 itself and 252 – but only one way of generating a value of, say, 240.) The NTRUEncrypt random polynomial generation constant *c*, is a value that is chosen for the deterministic generation of a polynomial from a pseudo-random number generator. It represents the number of bits that are used to generate a mod *N* entry candidate. The NTRUEncrypt random polynomial generation constant *c* is fixed for each degree *N*, and is chosen to minimize the number of octets expected to be output by the pseudo-random number generator. As an example, for *N* = 347, *c* is chosen to be 14 because 47*347 (=16309) is close to 2^{14} (=16384) so only the values greater than 16309 will be thrown out.

The specific value of *c* is defined for each parameter set listed in section 4.3.

3.1.3.4 NTRUEncrypt Message Length Encoding Length

For certain message padding methods, the length of the message that is to be encrypted is encoded in the padded message itself. When this type of message padding is used, the length of the field that represents the length of the message, called the message length encoding length, is represented by the parameter *lLen*. For parameter sets that require the length of the message to be less than 256 bytes, *lLen* is typically set to 1.

The specific value of *lLen* is defined for each parameter set listed in section 4.3.

3.1.3.5 Hash Function

NTRUEncrypt operations involve generating strings of pseudo-random output from a given input. The functions that generate this output are known as Mask Generation Functions, if they output a single arbitrary-length string, or Pseudo Random Number Generators, if they maintain state and produce output an arbitrary number of times. These two types of function are described in sections 3.1.3.6 and 3.1.3.7 below. Both of these functions are instantiated using a cryptographically strong hash function. The hash function shall be chosen from the set of approved hash functions listed in section 3.7.1. In this standard, the hash function used to instantiate the MGF must also be used to instantiate the PRNG.

In this standard, hash functions are considered to take octet strings as inputs and outputs.

3.1.3.6 Mask Generation Function

In NTRUEncrypt message representative generation methods, a Mask Generation Function (MGF) may be used to help compute the message representative from the message and random component. The MGF serves the purpose of making the output reasonably randomly distributed and making a single bit of the output rely on multiple bits of the input. An MGF is a construction built around a hash function. The hash function shall be chosen from the set of approved hash functions listed in section 3.7.1, and the MGF itself shall be chosen from the set of approved MGFs listed in section 3.7.2.

In this standard, Mask Generation Functions are considered to take octet strings as inputs and outputs.

3.1.3.7 Pseudo-Random Number Generation

In NTRUEncrypt blinding value generation methods, a pseudo-random number generation method (PRNG) may be used to help compute the blinding value. The pseudo-random number generation method serves the purpose of making an efficient and reasonably randomly distributed mapping from the message *m* and random component *b* to the blinding value. A PRNG is a construction built around a hash function. The hash function shall be chosen from the set of approved hash functions listed in section 3.7.1, and the PRNG shall be chosen from the set of approved PRNGs in section 3.7.3.

In this standard, PRNGs are considered to take octet strings as inputs and outputs.

3.1.3.8 Blinding Value Generation Method (BVGM)

In order to protect against chosen ciphertext attacks, NTRUEncrypt encryption is made plaintext-aware by using a deterministic blinding value generation method (BVGM). The BVGM may be used to compute the blinding value *r* from the padded message *pm*. In order to compute the same values, the entity performing encryption and the entity performing decryption shall use the same BVGM. The BVGM shall be chosen from the set of approved BVGM listed in section 3.3.1.

A BVGM takes as input the padded message *pm*, which is an octet string, and outputs the blinding value *r*, which is a ring element.

3.1.3.9 Key Generation Primitive (KGP)

In order to perform any operation in NTRUEncrypt, a key pair must be generated. NTRUEncrypt key generation primitives are used to create key pairs that satisfy the required security and efficiency properties. Once the key generation has been completed, the private key and public key should be retained by the party generating the key pair, and the public key may be distributed to the other parties. The KGP shall be chosen from the set of approved KGP listed in section 3.2.1.

3.1.3.10 Encryption Primitive (EP)

The basic operation performed during the encryption process using the public key is specified by the encryption primitive. Encryption primitives typically accept the message to be encrypted (called the *plaintext*) and the public key as input and return the encrypted message (called the *ciphertext*). The EP shall be chosen from the set of approved EP listed in section 3.2.2.

Note that the encryption scheme defined in this document does not actually use an encryption primitive. An encryption primitive is presented in this document for the sake of completeness.

3.1.3.11 Decryption Primitive (DP)

The basic operation performed during the decryption process using the private key is specified by the decryption primitive. Decryption primitives typically accept the encrypted message (ciphertext) and the private key as input and return a candidate message (plaintext) as output. The DP shall be chosen from the set of approved DP (and PDP) listed in section 3.2.3.

3.2 NTRUEncrypt Primitives

The following section defines the cryptographic primitives that are used in the NTRUEncrypt encryption scheme. These primitives include key generation primitives, encryption primitives and decryption primitives.

3.2.1 NTRUEncrypt Key Generation Primitives

For a given set of NTRUEncrypt domain parameters, an *NTRUEncrypt key pair* consists of an *NTRUEncrypt private key f*, which is a polynomial of degree N-1 chosen from the NTRUEncrypt private key space and an *NTRUEncrypt public key h*, which is a polynomial of degree N-1 equal to $f^{-1}*g*p$ modulo q.

NTRUEncrypt key pairs are closely associated with their domain parameters, and may only be used in the context of the domain parameters. A key pair shall not be used with a set of domain parameters different from the one for which it was generated. A set of domain parameters may be shared by a number of key pairs.

In this standard, the private key f: is taken to be of the form $f = 1 + pF$, where p is one of the domain parameters defined above, and F is a polynomial in the space D_F. The specific recommendations for the space D_F are given in section 4.3.

3.2.1.1 Random NTRUEncrypt Key Generation Primitive – KGP-NTRU1

An NTRUEncrypt key pair with f of the form $f = 1+pF$ may be generated using the following steps. Note that the algorithm below outputs only the values f, F and h. In some applications it may be desirable to store the values f^{-1} and g as well.

NTRUEncrypt Components:

— The NTRUEncrypt domain parameters N, q, p, df, dg

Input: None

Output: An NTRUEncrypt key pair consisting of the private key f and the public key h

Operation: The NTRUEncrypt key pair shall be computed by the following or an equivalent sequence of steps:

1. Randomly choose a polynomial F of degree N – 1 with df coefficients equal to 1 and the remaining coefficients equal to 0.

2. Compute the polynomial $f := 1 + p*F$ in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$
3. Compute the polynomial f^{-1} (i.e. the polynomial f^{-1} such that $f^{-1}*f = f* f^{-1} = 1$) in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$. If f^{-1} does not exist, go to step 1.
4. Randomly choose a polynomial g of degree $N - 1$ with dg coefficients equal to 1 and the remaining coefficients equal to 0.
5. Compute the polynomial g^{-1} (i.e. the polynomial g^{-1} such that $g^{-1}*g = g* g^{-1} = 1$) in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$. If g^{-1} does not exist, go to step 4.
6. Compute the polynomial $h := f^{-1}*g*p$ in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$
7. Output f, h and (optionally) F.

3.2.2 NTRUEncrypt Encryption Primitives

There is currently only one encryption primitive specified for NTRUEncrypt. The encryption primitive is the fundamental building block for the encryption operation. Note that the encryption scheme presented in section 4.2 does not make direct use of this primitive; it is presented for completeness only.

3.2.2.1 NTRUEncrypt Encryption Primitive – SVEP-NTRU

SVEP-NTRU is the NTRUEncrypt Encryption Primitive. It is based on the work of [HPS98] and [HS00-2]. It is invoked in the scheme SVES as part of encrypting a message, given the message representative and the public key of the intended recipient. The message can be decrypted within a scheme by invoking SVDP-NTRU.

NTRUEncrypt Components:
— The NTRUEncrypt parameters N, q

Input:
— The recipient's NTRUEncrypt public key h
— The message representative, which is a polynomial i
— The message blinding value, which is a polynomial r

Output: The encrypted message representative, which is a polynomial e

Operation: The encrypted message representative e shall be computed by the following or an equivalent sequence of steps:

1. Compute the polynomial $e := r*h + i$ in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$.
2. Output e.

Conformance region recommendation. A conformance region should include:
— At least one valid NTRUEncrypt public key h
— All message representatives i (and corresponding blinding values r, which are determined from i)

3.2.3 NTRUEncrypt Decryption Primitives

There is currently only one decryption primitive specified for use.

3.2.4 NTRUEncrypt Decryption Primitive: SVDP-NTRU2

Components:
— The NTRUEncrypt parameters N, q, p

Inputs:

— The recipient's NTRUEncrypt private key f, f^{-1} mod p.
— The encrypted message representative, which is a polynomial e.

Output:
— The candidate decrypted polynomial ci.

Operations: The candidate decrypted polynomial ci shall be calculated by the following or an equivalent sequence of steps:

1. Compute the polynomial $a := e*f$ in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$.
2. Compute the quantity $I := (a(1) - p(1)*r(1)*\mathrm{g}(1))*f(1)^{-1}$ (mod q), choosing I in the range $(N - q)/2 \le I < (N + q)/2$.
3. Compute the quantity $A :=$ floor[$(1/N) * (p(1)*r(1)*g(1) + I*f(1)) + N/2$] – ceil[$q/2$].
4. Compute the partially decrypted polynomial $a = f * e$ mod q, placing the coefficients of a into the range $[A+1, A+q]$.
5. Compute ci as

   $ci = f^{-1}*a$ mod p.

Notes:
For the parameter sets given in this document, f^{-1} mod p is equal to 1. In this case, step 5 becomes simply reduction mod p.

3.3 NTRUEncrypt Encoding Methods

Before a message is encrypted, it must be processed to guarantee certain desirable security properties such as semantic security. This processing typically involves the following steps:
— Adding random data to obtain the padded message
— "Masking" the padded message to obtain the message representative
— Deriving the blinding value from the padded message.

In this document the first two of the above steps are specified in the description of the SVES-3 encryption scheme. This section specifies a method for generating the blinding value r.

3.3.1 NTRUEncrypt Blinding Value Generation Methods

In order to provide plaintext awareness, a blinding value generation method (BVGM) shall be used to generate a blinding value r from a seed *seed*. This section contains the single BVGM approved for use with the parameter sets in this document. The BVGM generates a pseudo-random binary blinding value r.

3.3.1.1 Blinding Value Generation From *dr* – BVGM-NTRU1

The blinding value r shall be generated deterministically from a seed using a pseudo-random number generator. The precise form of this seed is defined by the encryption scheme, but it will at a minimum include the message m and the random component b. Note that in this standard the number of calls made to the PRNG may vary.

NTRUEncrypt Components:
— The NTRUEncrypt parameters N , dr
— The chosen pseudo-random number generator PRNG( )
— The hash function Hash() chosen to parameterize PRNG( )

— The random polynomial generation constant *c*

Input:
— The seed, which is an octet string *seed*

Output:
— The blinding value, which is a polynomial *r*

Operation: The blinding value shall be computed by the following or an equivalent sequence of steps:

```
1. Instantiate the pseudo-random number generator with hash function Hash() and input seed to
   produce an output stream PRNG (seed).
2. Set $B$ := ceil[$c$/8]
3. Set $t$ := 0
4. Set $r$ := 0
5. While $t < dr$ do
     a. Set $o$ := next $B$ octets of PRNG (seed)
     b. Set the high-order $8B - c$ bits of $o$ to 0
     c. Set $i$ := $o$ converted to an integer using OS2IP
     d. If $i < 2^c - (2^c \bmod N)$ and $r_{(i \bmod N)} = 0$
          i. Set $r_{(i \bmod N)}$ := 1
          ii. Set $t := t + 1$
6. Return $r$.
```

3.4 NTRUSign Components

This section defines the NTRUSign components, categorizes them (as domain parameters, security parameters or scheme options) and gives a basic description of the component. Security considerations for specific choices for the component are included when appropriate, however for detailed security considerations, see [IEEE P1363.1].

Instantiations of primitives and encoding methods (which are both scheme options) are specified in section 3.5 and 3.6 respectively. Required choices for all NTRUSign components are listed in section 5.3.

3.4.1 NTRUSign Domain Parameters

Values for each of the domain parameters must be selected in order to define the space in which operations are performed in NTRUSign. The domain parameters specified in the standard maximize efficiency and security.

3.4.1.1 NTRUSign Degree

The degree *N* identifies the dimension of the convolution polynomial ring used. Although *N* is referred to here as the NTRUSign degree, elements of the ring are represented as polynomials of degree $N - 1$.

The specific value of *N* is defined for each parameter set listed in section 5.3.

3.4.1.2 NTRUSign Big Modulus

As described in section 2.2.5, the big modulus *q* is used to define the polynomial ring used for NTRUSign. In this standard, the modulus *q* is chosen to be a power of 2.

The modulus q is defined for each parameter set listed in section 5.3.

3.4.2 NTRUSign Security Parameters

Values for each of the security parameters may be globally specified of chosen by the holder of the private key. These values must be chosen from those specified in the standard and need not be kept secret. The security parameters specified in the standard are selected to maximize efficiency and security.

3.4.2.1 NTRUSign Private Key Space

The NTRUSign private key consists of four polynomials (f, g, F, G). The polynomials f and g uniquely determine the private key. In general, of the four polynomials (f, g, F, G), only two are needed for signing. We denote these two by (f, f'). Depending on the basis type (see section 3.4.2.3), f' may be F (in the standard basis) or g (in the transpose basis). The two components of the private key which are not used for signing may be discarded after key generation.

Generally, the private key part f may be chosen to be any small polynomial f such that f is invertible in space $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$. However, for improved efficiency, f is chosen in this standard to be in the space D_f of all polynomials of degree $N - 1$ that have df coefficients equal to 1 and the remaining coefficients equal to 0.

Generally, the private key part g may be chosen to be any small polynomial g. However, for improved efficiency, g is chosen in this standard to be in the space D_g of all polynomials of degree $N - 1$ that have dg coefficients equal to 1 and the remaining coefficients equal to 0.

In addition, for certain key generation methods, it is necessary that the resultants of f and g with respect to $X^N - 1$ are relatively prime. As a result, it is important to choose the parameters df and dg to be relatively prime to allow this to occur.

Once f and g have been chosen, the basis completion pair (F, G) is computed such that (f, g) and (F, G) form a small basis for the NTRUSign module – in other words, such that $fG - Fg = q$ and (F, G) is small (typically of size about $\sqrt{(N/12)}$ times the size of (f, g)). The basis completion pair is not unique for a given (f, g), but only one such pair is needed.

Generally, the basis completion pair may be any pair that completes the basis. However, in order to maximize the probability of generating good signatures, the basis completion pair may be restricted to the space D_{FG} that consists of all vectors whose centered norm is smaller than the security parameter *KeyNormBound*. If during key generation, a basis completion pair cannot be found with centered norm less than *KeyNormBound*, the private key may be discarded and another private key chosen.

Recommended values of df and dg are included with each parameter set listed in section 5.3. These parameters are selected to provide maximum security and efficiency.

The parameter *KeyNormBound* is not used in any of the key generation techniques currently described in the standard.

3.4.2.2 NTRUSign Basis Completion Maximum Adjustment

When computing the basis completion pair (F, G) during key generation, it is possible to decrease the centered norm of the basis completion pair (F, G) by adding or subtracting small multiples of the private key pair (f, g). After several iterations, the loss in key generation efficiency may outweigh the gain from a decreased centered norm for (F, G), so the security parameter *MaxAdjustment* may be chosen to limit the iterations of this algorithm.

Recommended values for *MaxAdjustment* are included in each parameter set listed in section 5.3.

3.4.2.3 NTRUSign Private Key Type

An NTRUSign basis is a set of polynomials (f, g, F, G) such that $fG - Fg = q$. In the standard NTRU lattice, the basis consists of the two vectors (f, g) and (F, G) and all of their componentwise rotations. Each basis of this type also defines a *transpose* basis (f, F) and (g, G), also of determinant q. It may be significantly faster to use the transpose basis, rather than the standard basis, as the private key for signature generation. However, if the transpose basis is used, one component of the signature will be significantly smaller than the other. This will lead to transcripts converging faster for the transpose basis than for the standard basis. For this reason, the use of the transpose basis is only recommended if at least one perturbation basis (see section 3.4.2.5) is used when signing.

The type of basis used is given by the variable *basisType*, which can take the values “standard” or “transpose”. Recommended values for *basisType* are included in each parameter set listed in section 5.3.

3.4.2.4 NTRUSign Public Key Space

The NTRUSign public key space is uniquely determined by the NTRUSign private key space and hence need not be specified explicitly as a security parameter. In the standard lattice, the NTRUSign public key space D_h consists of all polynomials h of degree $N - 1$ with $h = f^{-1}*g$ with coefficients reduced modulo q, where f, g are chosen from D_f and D_g respectively and f^{-1} is the polynomial with coefficients reduced mod q such that $f^{-1}*f = f*f^{-1} = 1$ in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$. In the transpose basis, the public key space consists of all polynomials h of degree $N - 1$ with $h = f^{-1}*F \bmod q$.

3.4.2.5 NTRUSign Perturbation Bases

NTRUSign signatures are not zero knowledge, and a transcript of signatures will gradually reveal information about the private key, leading in extreme cases to recovery of the key by an attacker. This information leakage can be slowed down by the use of *perturbations*. This refers to a technique where the signer first signs the message using an entirely private basis, such that no public information at all is known about this basis, to produce a *perturbed message point* close to the original message point. The signer then signs the perturbed message point with the real private key to produce a signature that can be verified using the public key.

This process can clearly be extended to cover the use of multiple perturbation bases, each of which acts on the previous perturbed message point to produce a perturbed message point of its own. The final perturbed message point in the secret can then be signed to produce the signature.

Each perturbation basis used greatly increases the number of signatures that an attacker needs to mount an attack on the private key. However, the use of perturbation bases also increases the average signature norm. On average, use of B bases will increase the norms of signatures by $\sqrt{(B+1)}$.

The number of perturbation bases used is given by the variable *perturbationBases*. Recommended values for *perturbationBases* are included in each parameter set listed in section 5.3. Note that if the standard lattice is being used, the perturbation bases will also be in standard form, and if the transpose lattice is being used, the perturbation bases will also be in transpose form.

3.4.2.6 NTRUSign Signature Failure Tolerance

Depending on the size of the norm bound for signatures, the signature primitive may occasionally fail to produce a valid signature on a given message m with a given message randomization value r. During the signing process, the signer may choose another message randomization value to produce a different message representative and a different signature. Due to efficiency reasons, it may be desirable to simply fail the signature if after a certain number of attempts, a valid signature cannot be found. The signature failure tolerance *SignFailTolerance* is chosen to specify the number of attempts made before the signature process returns a failed signature. Note that in general this number may be set to a very small number as most signature attempts will pass.

The specific value of the *SignFailTolerance* is variable for each parameter set listed in section 5.3., however recommended choices are specified.

3.4.3 NTRUSign Scheme Options

NTRUSign scheme options consist of parameters and algorithms that do not affect the key space (e.g. that are not domain parameters), but that must be agreed upon in order to implement the NTRUSign signature scheme. Scheme options include the chosen primitives and encoding methods and the parameters that are needed to completely specify the encoding methods and primitives.

3.4.3.1 NTRUSign Signature Norm Bound

The NTRUSign signature norm bound *NormBound* is chosen to indicate how close the signature must be to the message representative in order for signatures to verify. It is selected to be small enough to prevent forgery attacks and large enough to make the probability of the signature being below the norm bound high.

The specific value of the *NormBound* is defined for each parameter set listed in section 5.3.

3.4.3.2 NTRUSign Message Randomization Value

The NTRUSign signature algorithm deterministically finds a reasonably close NTRUSign lattice point to the message representative. Due to the unpredictable nature of the closeness of these lattice points, a signature attempt may on rare occasions fail to satisfy the needed norm bound. The NTRUSign message randomization value r is used to generate a different message representative should the signature fail to satisfy the norm bound.

The message randomization value is chosen from the message randomization value space D_r. The message randomization value space is chosen to be any octet string of length *lr*. If the signature is expected never to fail this test, the value of *lr* may be set to 0 and hence, the message randomization value may be omitted. Note that a fixed r may be chosen for all signature attempts and not be communicated to the verifier along with the signature.

The specific values of *lr* and r are defined for each parameter set in section 5.3.

3.4.3.3 NTRUSign Random Polynomial Generation Constant

When randomly generating a polynomial from a pseudo-random number generator, such as in certain message representative generation methods, the number of output bits used to compute the next polynomial value needs to be selected. The NTRUSign random polynomial generation constant c is a value that is chosen for the deterministic generation of a polynomial from a pseudo-random number generator. It represents the number of bits that are used to generate a mod N entry candidate. The NTRUSign random polynomial generation constant c is fixed for each degree N and is chosen to minimize the number of octets needed to compute the polynomial. Note that in certain message representative generation methods, the polynomial is chosen as mod q coefficients (which is an even power of 2) instead of selecting the coefficient location mod N. Therefore c is not needed for those methods.

The specific value of c is defined for each parameter set listed in section 5.3.

3.4.3.4 Hash Function

In NTRUSign message representative generation methods, a hash function is used on the message to establish the seed for the pseudo-random number generator. The hash function serves the purpose of making the input to the PRNG small while still maintaining the feature that each input bit affects each output bit. The hash function may also be used as a part of the PRNG. The hash function shall be chosen from the set of approved hash functions listed in section 3.7.1 and is defined for each parameter set in section 5.3.

In this standard, hash functions are considered to take octet strings as inputs and outputs.

3.4.3.5 Pseudo-Random Number Generation

In NTRUSign message representative generation methods, a pseudo-random number generation method may be used to help compute the message representative. The PRNG serves the purpose of making an efficient and reasonably randomly distributed mapping from the hashed message to the message representative. The PRNG shall be chosen from the set of approved PRNG in section 3.7.3 and is defined for each parameter set in section 5.3.

In this standard, PRNGs are considered to take octet strings as inputs and outputs.

3.4.3.6 NTRUSign Message Representative Generation Method (MRGM)

In order to protect against two messages being represented by points that are close to each other in the NTRUSign lattice, NTRUSign message representative generation methods (MRGM) are used to transform the messages *m* into message representatives *i* that are reasonably evenly distributed through the NTRUSign message space. In order to compute the same values, the entity performing the signing and the entity performing the verification shall use the same MRGM. The MRGM shall be chosen from the set of approved MRGM listed in section 3.6.1 and is defined for each parameter set in section 5.3.

A MRGM takes as input the message *m*, which is an octet string, and outputs the message representative *i*, which is a ring element.

3.4.3.7 NTRUSign Key Generation Primitive (KGP)

In order to perform any operation in NTRUSign, a key pair must be generated. NTRUSign key generation primitives are used to create key pairs and basis completion pairs that satisfy the required security and efficiency properties. Once the key generation has been completed, the private key and public key should be retained by the party generating the key pair and the public key should be distributed to the party that will be performing the signature verification with the public key.

The KGP shall be chosen from the set of approved KGP listed in section 3.5.1 and is defined for each parameter set in section 5.3.

3.4.3.8 NTRUSign Signature Primitive (SP)

The basic operation performed during the signing process using the private key is specified by the signature primitive. Signature primitives typically accept the message representative and the private key as input and return the signature.

The SP shall be chosen from the set of approved signature primitives listed in section 3.5.2 and is defined for each parameter set in section 5.3.

3.4.3.9 NTRUSign Verification Primitive (VP)

The basic operation performed during signature verification using the public key is specified by the verification primitive. Verification primitives typically accept the message representative, the signature and the public key as input and return "valid" or "invalid".

The VP shall be chosen from the set of approved verification primitives listed in section 3.5.3 and is defined for each parameter set in section 5.3.

3.5 NTRUSign Primitives

3.5.1 NTRUSign Key Generation Primitives

For a given set of NTRUSign domain parameters, an *NTRUSign key pair* consists of an *NTRUSign private key basis (f, f')*, an *NTRUSign public key h*, which is a polynomial of degree N-1 equal to $f^{-1}*f'$ modulo q, and zero or more *perturbation bases* (f_i, f'_i, h_i), where $h_i = f_i^{-1}*f'_i$ modulo q.

NTRUSign key pairs are closely associated with their domain parameters, and shall only be used in the context of the domain parameters. A key pair shall not be used with a set of domain parameters different from the one for which it was generated. A set of domain parameters may be shared by a number of key pairs.

Key generation for NTRUSign consists of generating one or more distinct NTRUSign module bases. For clarity, we present key generation here in two steps: first, an algorithm for generating a random NTRUSign basis; second, a key generation primitive which uses the basis generation algorithm to generate a complete private key.

3.5.1.1 Random NTRUSign Basis Generation

An NTRUSign basis may be generated using the following steps. The optional step 3 in the algorithm below is included to efficiently determine if the resultants of f and g are both divisible by $2N + 1$, which is the most likely common factor of the resultants. This prevents performing the expensive full resultant calculations on an (f, g) pair that will be discarded in step 7 anyway. The optional steps 21-27 are included to further reduce the size of the basis completion pair (F, G) by adding (or subtracting) rotations of f and g. The smaller the basis completion pair is, the smaller (on average) the norms of the signatures will be.

NTRU Components:
— The NTRUSign domain parameters N, q
— The NTRUSign key security parameters *df, dg, MaxAdjustment* (optional)

Input: None

Output: An NTRUSign basis consisting of the polynomials (f, g, F, G).

Operation: The NTRUSign basis shall be computed by the following or an equivalent sequence of steps:

1. Randomly choose a polynomial f of degree $N – 1$ with *df* coefficients equal to 1 and the remaining coefficients equal to 0.
2. Randomly choose a polynomial g of degree $N – 1$ with *dg* coefficients equal to 1 and the remaining coefficients equal to 0.
3. (optional) If $2N + 1$ is prime

```
        a. Set integers $resf1$ and $resg1$ equal to the resultants of $f$ and $g$ (mod $2N + 1$) respectively,
           calculating the resultants using algorithm 2.2.7.1
        b. If $resf1$ and $resg1$ are both 0, go to step 1
 4. Set integer $resf$ equal to the resultant of $f$ and set the polynomial $rhof$ to satisfy the equation $rhof*f$
    $:= resf$ in $\mathbf{Z}[X]/(X^N - 1)$, using algorithm 2.2.7.2
 5. Set integer $resg$ equal to the resultant of $g$ and set the polynomials $rhog$ to satisfy the equation
    $rhog*g := resg$ in $\mathbf{Z}[X]/(X^N - 1)$, using algorithm 2.2.7.2
 6. Compute integers $alpha$, $beta$ and the gcd $gcd$ of $resf$ and $resg$ such that $alpha*resf + beta*resg =$
    $gcd$ (using the Extended Euclidean Algorithm in the integers)
 7. If $gcd$ is not equal to 1, go to step 1
 8. Compute the polynomial $f^{-1}$ (i.e. the polynomial $f^{-1}$ such that $f^{-1}*f = f*f^{-1} = 1$) in
    $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$. If $f^{-1}$ does not exist, go to step 1.
 9. Set polynomial $F := -rhog*beta*q$ in $\mathbf{Z}[X]/(X^N - 1)$
10. Set polynomial $G := rhof*alpha*q$ in $\mathbf{Z}[X]/(X^N - 1)$
11. Let $frev$ be the polynomial of degree $N - 1$ such that $frev_0 = f_0$ and $frev_i = f_{N-i}$ for $1 \le i \le N - 1$
    (this polynomial is called the reversal of $f$)
12. Let $grev$ be the polynomial of degree $N - 1$ such that $grev_0 = g_0$ and $grev_i = g_{N-i}$ for $1 \le i \le N - 1$
    (this polynomial is called the reversal of $g$)
13. Set polynomial $t := f*frev + g*grev$ in $\mathbf{Z}[X]/(X^N - 1)$
14. Set integer $rest$ equal to the resultant of $t$ and set the polynomial $rhot$ to satisfy the equation $rhot*t$
    $:= rest$ in $\mathbf{Z}[X]/(X^N - 1)$, using algorithm 2.2.7.2
15. Set polynomial $c := rhot*(frev*F + grev*G)$ in $\mathbf{Z}[X]/(X^N - 1)$
16. Set integers $i := 0$, $j := 0$, $k := 0$
17. While $j < N$ do
        a. Set $c_j := \text{floor}[c_j/rest + .5]$
        b. Set $j := j + 1$
18. Set $F := F - c*f$ in $\mathbf{Z}[X]/(X^N - 1)$
19. Set $G := G - c*g$ in $\mathbf{Z}[X]/(X^N - 1)$
20. (optional) Set integers $D := 0$, $E := 0$
21. (optional) Set polynomials $u := f$, $v := g$
22. (optional) Set $j := 0$
23. (optional) While $j < N$ do
        a. Set $E := E + 2*N*(f_j^2 + g_j^2)$
        b. Set $j := j + 1$
24. (optional) Set $E := E - (f(1) + g(1))^2$
25. (optional) Set $j := 0$
26. (optional) While $k < MaxAdjustment$ and $j < N$ do
        a. Set $D := 0$
        b. While $i < N$ do
               i. Set $D := D + 4*N*(F_i*f_i + G_i*g_i)$
              ii. Set $i := i + 1$
        c. Set $D := D - 2*(F(1) + G(1))*(f(1) + g(1))$
        d. If $D > E$
               i. Set $F := F - u$ in $\mathbf{Z}[X]/(X^N - 1)$
              ii. Set $G := G - v$ in $\mathbf{Z}[X]/(X^N - 1)$
             iii. Set $k := k + 1$
              iv. Set $j := 0$
        e. Else, if $D < -E$
               i. Set $F := F + u$ in $\mathbf{Z}[X]/(X^N - 1)$
              ii. Set $G := G + v$ in $\mathbf{Z}[X]/(X^N - 1)$
             iii. Set $k := k + 1$
              iv. Set $j := 0$
        f. Set $j := j + 1$
        g. Set $u := u*X$ in $\mathbf{Z}[X]/(X^N - 1)$
        h. Set $v := v*X$ in $\mathbf{Z}[X]/(X^N - 1)$
27. Output $f$, $g$, $F$, $G$.
```

3.5.1.2 NTRUSign Key Generation Primitive – KGP-NTRUSign1

The following primitive outputs an NTRUSign keypair.

NTRU Components:
— The NTRUSign domain parameters *N*, *q*
— The NTRUSign key security parameters *df*, *dg*, *MaxAdjustment* (optional)
— The *basisType* “standard” or “transpose”
— The number of perturbation bases, *perturbationBases*

Input: None

Output: An NTRUSign private key consisting of the private key polynomials (f, f'_i) and *perturbationBases* number of private perturbation bases (f_i, f'_i, h_i), and the NTRUSign public key h.

Operation: The NTRUSign keypair shall be computed by the following or an equivalent sequence of steps:

```
1. Set $i$ = perturbationBases.
2. While $i \geq 0$:
    a. Generate an NTRUSign basis ($f_i$, $g_i$, $F_i$, $G_i$) using the algorithm given in section 3.5.1.1.
    b. If basisType = “standard”, set $f'_i = F_i$. If basisType = “transpose”, set $f'_i = g_i$. Set $h_i = f_i^{-1} * f'_i \bmod q$.
    c. Set $i = i-1$.
3. The public key is $h_0$. The private key is the set ($f_i$, $f'_i$, $h_i$) for $0 \leq i \leq$ perturbationBases.
```

3.5.2 NTRUSign Signature Primitives

The NTRUSign signature primitives are used to generate a secure digital signature from a message representative. There is only one NTRUSign signature primitive specified in this standard.

3.5.2.1 NTRUSign Signature Primitive – SVSP-NTRU

SVSP-NTRU is the NTRU Signature Primitive. It is based on the work of [HHPSW01]. SVSP-NTRU may be used in a signature scheme with appendix and can be invoked in the scheme SVSSA as part of signature generation. Note that the message representative i may be the product of multiple small components (e.g. $i = i0*i1*\ldots*ij$ in $\mathbf{Z}[X]/(X^N - 1)$).

NTRUSign Components:
— The NTRUSign domain parameters *N*, *q*

Input:
— The signer’s NTRUSign private key (f, f')
— The signer’s NTRUSign perturbation bases (f_i, f'_i, h_i) and the number *perturbationBases*
— The message representative, which is a polynomial *i*

Output: The signature, which is a polynomial *s*

Operation: The signature *s* shall be computed by the following or an equivalent sequence of steps:

```
1. Set $s = 0$. Set iLoop = perturbationBases.
2. While $iLoop \geq 1$:
    a. Compute the polynomial $B = -f_{iLoop}'*i$ in $\mathbf{Z}[X]/(X^N - 1)$ (only need to store $2*\log_2 q$ bits per coefficient)
    b. Set integer $j := 0$
```

```
        c. While $j < N$ do
            i. Set $B_j := \text{floor}[B_j/q + .5]$
            ii. Set $j := j + 1$
        d. Compute the polynomial $b = f_{iLoop}*i$ in $\mathbf{Z}[X]/(X^N - 1)$ (only need to store $2\log_2 q$ bits per coefficient)
        e. Set $j := 0$
        f. While $j < N$ do
            i. Set $b_j := \text{floor}[b_j/q + .5]$
            ii. Set $j := j + 1$
        g. Set polynomial $s_{iLoop} := b* f_{iLoop}' + B*f_{iLoop}$ in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$
        h. Set $s := s + s_{iLoop} \bmod q$. Set $i := s_{iLoop} * (h_{iLoop} - h_{iLoop+1}) \bmod q$. Set $iLoop := iLoop - 1$.
3.  Compute the polynomial $B = -f*i$ in $\mathbf{Z}[X]/(X^N - 1)$
4.  Set integer $j := 0$
5.  While $j < N$ do
        a. Set $B_j := \text{floor}[B_j/q + .5]$
        b. Set $j := j + 1$
6.  Compute the polynomial $b = f*i$ in $\mathbf{Z}[X]/(X^N - 1)$
7.  Set $j := 0$
8.  While $j < N$ do
        a. Set $b_j := \text{floor}[b_j/q + .5]$
        b. Set $j := j + 1$
9.  Set polynomial $s_0 := b* f' + B*f$ in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$
10. Output $s := s + s_0 \bmod q$.
```

3.5.3 NTRUSign Verification Primitives

The NTRUSign verification primitives are used to indicate if a signature on a message representative satisfies the appropriate verification conditions or not. There is only one NTRUSign verification primitive specified in this standard.

3.5.3.1 NTRUSign Verification Primitive – SVVP-NTRU

SVVP-NTRU is the NTRU Verification Primitive. It is based on the work of [HHPSW01]. SVVP-NTRU may be used in a signature scheme with appendix and can be invoked in the scheme SVSSA as part of signature verification. Note that the message representative i may be the product of multiple small components (e.g. $i = i0*i1*\ldots*ij$ in $\mathbf{Z}[X]/(X^N - 1)$).

NTRUSign Components:
— The NTRUSign parameters N, q
— The NTRUSign security parameter *NormBound*

Input:
— The signer's NTRUSign public key h
— The signature to be verified, which is a polynomial s
— The message representative i for which s is alleged to be a signature

Output: A message indicating that the signature is either "valid" or "invalid"

Operation: A signature s shall be verified by the following or an equivalent sequence of steps:

```
1. Compute the polynomial $t := h*s$ in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$
2. Compute the polynomial $e2 := i - t$ in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$ (setting coefficients in the range 0 to $q - 1$)
```

```
3.  Let $maxrange$ be the largest integer such that $e2_j – e2_k = maxrange$ for some $j, k$ in the range 0 to $q – 1$ and no coefficient of $e2$ has values between $e2_j$ and $e2_k$
4.  Let $e2_l$ be the largest coefficient of $e2$ and $e2_m$ be the smallest coefficient of $e2$
5.  Set integer $j := q – e2_l + e2_m$
6.  If $j > maxrange$
        a. Set integer $shift := m$
7.  Else
        a. Set integer $shift := j$
8.  Set $j := 0$
9.  While $j < N$ do
        a. Set $e2_j := e2_j – shift \pmod q$
        b. Set $j := j + 1$
10. Let $maxrange$ be the largest integer such that $s_j – s_k = maxrange$ for some $j, k$ in the range 0 to $q – 1$ and no coefficient of $s$ has values between $s_j$ and $s_k$
11. Let $s_l$ be the largest coefficient of $s$ and $s_m$ be the smallest coefficient of $s$
12. Set $j := q – s_l + s_m$
13. If $j > maxrange$
        a. Set $shift := m$
14. Else
        a. Set $shift := j$
15. Set $j := 0$
16. While $j < N$ do
        a. Set $s_j := s_j – shift \pmod q$
        b. Set $j := j + 1$
17. Set $j := 0$
18. Set integers $ssum, e2sum, squaresum := 0$
19. While $j < N$ do
        a. Set $ssum := ssum + s_j$
        b. Set $e2sum := e2sum + e2_j$
        c. Set $squaresum := squaresum + s_j^2 + e2_j^2$
        d. Set $j := j + 1$
20. Compute the value $CenteredNorm := \mathrm{sqrt}((N*squaresum – ssum^2 – e2sum^2)/N)$
21. If $CenteredNorm > NormBound$
        a. Output "invalid"
22. Else
        a. Output "valid"
```

3.6 NTRUSign Encoding Methods

3.6.1 NTRUSign Message Representative Generation Methods

An MRGM must have the property that it should be computationally infeasible to find two messages whose message representatives are close to each other.

When signing a message, the signature shall be applied to the message representative *i*, which is the result of a one-way operation on the message. In order to generate the message representative, a hash function shall be applied to the message *m* and the resulting value is used to generate the message representative polynomial *i*.

The message representative *i* represents the point in $\mathbf{Z}^{2N}$ $[0^N, i]$, where 0^N represents the N-tuple of all 0's and *i* represents the N-tuple with entries equal to the coefficient values i_j of the polynomial *i*.

3.6.1.1 Message Representative Generation – MRGM-NTRUSign1

The following message representative generation method is a supported method for NTRUSign. This method consists of the generation of a single pseudo-random polynomial with coefficients in the range $[0, q – 1]$. If this method is to be used, q should be a power of 2.

NTRUSign Components:
— The NTRUSign parameters N, q, where q is a power of 2.
— The NTRUSign security parameter lr
— The chosen pseudo-random number generator PRNG( )
— The chosen message hash function Hash( )
— The hash function used to instantiate the PRNG PrngHash( )

Input:
— The message, which is an octet string m
— The message randomization value, which is an octet string r of length lr (may be the empty string)

Output: The message representative polynomial i

Operation: The message representative i shall be produced by the following or an equivalent sequence of steps:

```
1. Set $c := \log_2 q$ (i.e. the number of bits in $q$, e.g. if $q = 128$, $c = 7$)
2. Set $B := \mathrm{ceil}[c/8]$
3. Use the specified message hash function with input $m||r$ to produce an output Hash($m||r$).
4. Instantiate the pseudo-random number generator with hash function PrngHash() and input
   Hash($m||r$) to produce an output stream PRNG (Hash($m||r$)).
5. Set $t := 0$
6. While $t < N$ do
   a. Set $o :=$ next $B$ octets of PRNG (Hash($m||r$))
   b. Set the high-order $8B - c$ bits of $o$ to 0
   c. Set $j := o$ converted to an integer using OS2IP
   d. Set $i_t := j$
   e. Set $t := t + 1$
7. Return $i$
```

3.6.1.2 Message Representative Generation – MRGM-NTRUSign2

The following message representative generation method is a supported method for NTRUSign. This method includes the generation of multiple small message components *i0, i1, i2 … i(NumGroups – 1)* that are multiplied together to produce the message representative *i*. Note that it is usually desirable to store the message components in place of the message representative for computational efficiencies.

NTRUSign Components:
— The NTRUSign parameters N, q
— The NTRUSign security parameter lr
— The NTRUSign MRGM parameters *NumGroups*, *NumElements*
— The random polynomial generation constant c
— The chosen pseudo-random number generator PRNG( )
— The chosen hash function Hash( )

Input:

— The message, which is a bit string *m*
— The message randomization value *r* of length *lr* (may be the empty string)

Output: The message representative polynomial *i* (and optionally the message components *i0*, *i1*, …, *i(NumGroups – 1)*)

Operation: The message representative *i* shall be produced by the following or an equivalent sequence of steps:

```
1. Set $B$ := ceil[$c$/8]
2. Use the specified hash function with input $m||r$ to produce an output Hash($m||r$).
3. Instantiate the pseudo-random number generator with hash function PrngHash() and input
   Hash($m||r$) to produce an output stream PRNG (Hash($m||r$)).
4. Set $t$ := 0
5. Set $temp$ := 0
6. While $t < NumGroups$ do
   a. Set polynomial $it$ := 0 (e.g. for $t$ = 0, set polynomial $i0$ equal to 0, for $t$ = 1, set
      polynomial $i1$ equal to 0, etc.)
   b. Set $it_0 := it_0 + 1$
   c. While $temp < NumElements – 1$ do
      i. Set $o$ := next $B$ octets of PRNG (Hash($m||r$))
      ii. Set the leftmost $8B - c$ bits of $o$ to 0
      iii. Set $j$ := $o$ converted to an integer using OS2IP
      iv. If $j < 2^c – (2^c \bmod N)$
         1. Set $it_{(j \bmod N)} := it_{(j \bmod N)} + 1$
         2. Set $temp := temp + 1$
   d. Set $temp$ := 0
   e. Set $t := t + 1$
7. Set $i := i0*i1*i2*…*i(NumGroups – 1)$ in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N – 1)$
8. Return $i$ (and optionally $i0, i1, …, i(NumGroups – 1)$)
```

3.7 Supporting Algorithms

In order to perform the NTRUEncrypt operations securely, implementers shall choose supporting algorithms that satisfy the security needs of the schemes. The security level of the supporting algorithm typically depends on the desired security level of the scheme (e.g. for a desired security level of 80 bits, the SHA-1 hash algorithm is typically chosen). This section defines the algorithms that shall be used to meet this standard.

3.7.1 Hash Functions

Hash functions are used in two distinct situations in this standard: first, to hash a message before signing; second, as the core of a mask generation function. For security purposes, the hash function should be chosen at a strength commensurate to the desired security level. Note that the security requirements in the first case may be different from the security requirements in the second. The recommended parameter sets in this document specify hash functions appropriate to their security levels.

The only currently supported hash functions for hashing a message before signing are SHA-1, SHA-256, SHA-384 and SHA-512 [FIP95, NIST-SHA-2].

The only currently supported hash functions for use within a mask generation function are SHA-1, SHA-256, SHA-384, SHA-512 [FIP95, NIST-SHA-2] and MDC-2DES-NTRU, specified below.

All hash functions in this standard take an octet string as an input and produce an octet string as an output. For compatibility with other standards which specify input and output as bit strings, the conversion primitives OS2BSP and BS2OSP may be used.

3.7.1.1 Hash Function – MDC-2DES-NTRU

The following hash function, based on the Matyas-Meyer-Oseas MDC construction, uses DES encryption with two changing keys to generate a message digest of length 8 octets.

A DES key has 56 cryptographically significant bits, but is conventionally represented as a string of eight octets in which the rightmost bit of each octet is a parity bit. This standard follows this convention.

Input: The message, which is an octet string m of length l octets, with $l < 2^{32}$.

Output: The message digest, which is an octet string *md* of length 8 octets; or "error".

Operation: The message digest *md* shall be produced by the following or an equivalent sequence of steps:

1. If $l > 2^{32}$, output "error" and exit.
2. Set the initial keys K_0, K_0' to be the following octet string: K_0 := 52 52 52 52 52 52 52 52, K_0' := 25 25 25 25 25 25 25 25.
3. Set *NumZeroes* := 8 – ((l + 5) mod 8). If *NumZeroes* is equal to 8, set *NumZeroes* := 0.
4. Create the octet string *oZeroes*, consisting of *NumZeroes* octets with the value 00.
5. Create the octet string *oLen* by converting the number l to an octet string of length 4 using I2OSP.
6. Set the octet string m' := (m||80||*oZeroes*||*oLen*).
7. Set l' equal to the length in octets of m'. The octets in m' are indexed as $m'_0\ m'_1\ m'_2\ m'_3 \dots m'_{l'-1}$.
8. Set t:=0.
9. While $t < l'/8$ do
   a. Set encryption block B_t equal to the octet string $m'_{8t}\ m'_{8t+1}\ m'_{8t+2}\ m'_{8t+3} \dots m'_{8t+7}$.
   b. Calculate the intermediate ciphertext I_t := DesEncrypt (B_t, K_t)
   c. Calculate the ciphertext C_t := DesEncrypt (I_t, K_t')
   d. Calculate the intermediate hash $H_t := C_t$ XOR B_t.
   e. Set $K_{t+1} := H_t$.
   f. Counting from the left, set the second and third bits of K_{t+1} to 1 and 0 respectively.
   g. If necessary, set the parity bits of K_{t+1}.
   h. Get the initial value of K_{t+1}' from H_t by setting each octet of K_{t+1}' to the equivalent octet of H_t, right-rotated by four bits (for example, if the first octet of H_t in binary is 01011110, the first octet of K_{t+1}' is 11100101).
   i. Counting from the left, set the second and third bits of K_{t+1}' to 1 and 0 respectively.
   j. If necessary, set the parity bits of K_{t+1}'.
   k. Set $t := t+1$.
10. Set encryption block B_t equal to the octet string $m'_{l'-8}\ m'_{l'-7}\ m'_{l'-6} \dots m'_{l'-1}$.
11. Calculate the intermediate ciphertext I_t := DesEncrypt (B_t, K_t)
12. Calculate the ciphertext C_t := DesEncrypt (I_t, K_t')
13. Calculate the final hash $H_t := C_t$ XOR B_t.
14. Return H_t.

3.7.2 Mask Generation Functions

Mask Generation Functions (MGFs) are functions similar to hash functions, except that instead of producing a fixed-length output they produce an output of arbitrary length.

All mask generation functions are parameterized by the choice of a core hash function. The only hash functions supported for use with the MGFs in this standard are SHA-1, SHA-256, SHA-384, SHA-512 [FIP95, NIST-SHA-2] and MDC-2DES-NTRU, specified in section 3.7.1.1 of this document.

This standard permits the use of two mask generation functions: the MGF1, as specified in IEEE Standard 1363-2000 [IEEE 1363]; and the MGF-MDC-NTRU function, specified below.

All mask generation functions in this standard take as input an octet string and the desired length of the output, and output an octet string.

3.7.2.1 Mask Generation Function – MGF-1

This mask generation function is MGF-1 as specified in [IEEE 1363]. The only hash functions supported for use with this mask generation function are SHA-1, SHA-256, SHA-384, and SHA-512 [FIP95, NIST-SHA-2]

The function is parameterized by the following choice:
— A hash function *Hash* with output length *hLen* octets.

Input:
— An octet string Z of length *zLen* octets
— The desired length of the output, which is a positive integer *oLen*. (*oLen* shall be less than or equal to $hLen \times 2^{32}$).

Output: An octet string *mask* of length *oLen* octets; or “error”.

Operation: The octet string *mask* shall be produced by the following or an equivalent sequence of steps:

```
1. If $oLen$ exceeds $hLen \times 2^{32}$, or if $zLen$ exceeds any input length limitation on the hash function
   $Hash$, output “error” and exit.
2. Let $M$ be the empty string. Let $cThreshold$ = ceil[$oLen/hLen$].
3. Set $counter := 0$.
4. While $counter < cThreshold$ do
   a. Convert $counter$ to an octet string $C$ of length 4 octets using I2OSP.
   b. Compute $Hash(Z || C)$ with the selected hash function to produce an octet string $H$ of
      length $hLen$ octets.
   c. Let $M = M || H$.
   d. Increment $counter$ by one.
5. Output the leading $oLen$ octets of $M$ as the octet string $mask$.
```

3.7.2.2 Mask Generation Function – MGF-MDC-NTRU

This mask generation function is a variant of MGF1 above. It is designed for the case where the output length of the core hash function is short. The only hash function supported for use with this MGF is MDC-2DES-NTRU, specified in section 3.7.1.1 of this document.

The function is parameterized by the following choice:
— A hash function *Hash* with output length *hLen* octets.

Input:
— An octet string *Z* of length *zLen* octets
— The desired length of the output, which is a positive integer *oLen*. (*oLen* shall be less than or equal to $hLen \times 2^{64}$).

Output: An octet string *mask* of length *oLen* octets; or "error".

Operation: The octet string *mask* shall be produced by the following or an equivalent sequence of steps:

```
1. If oLen exceeds $hLen \times 2^{64}$, or if zLen exceeds any input length limitation on the hash function
   Hash, output "error" and exit.
2. Let M be the empty string. Let cThreshold = ceil[oLen/hLen].
3. Set counter := 0.
4. While counter < cThreshold do
   a. Convert counter to an octet string C of length 8 octets using I2OSP.
   b. Compute Hash(C || Z) with the selected hash function to produce an octet string H of
      length hLen octets.
   c. Let M = M || H.
   d. Increment counter by one.
5. Output the leading oLen octets of M as the octet string mask.
```

3.7.3 Pseudo-Random Number Generation

The term "pseudo-random number generators", as used in this standard, applies to functions which are initialized with an octet string and may then be called repeatedly, producing output of a specified but arbitrary length on each call. They differ from mask generation functions in that they may be called multiple times, while a mask generation function may only be called once.

All pseudo-random number generation functions are parameterized by the choice of a core hash function. The only hash functions supported for use with the MGFs in this standard are SHA-1, SHA-256, SHA-384, SHA-512 [FIP95, NIST-SHA-2] and MDC-2DES-NTRU, specified in section 3.7.1.1 of this document.

This standard permits the use of two random number generators: one based on MGF1, and one based on MGF-MDC-NTRU.

All random number generators in this standard are initialized with an octet string, and, when called, output an octet string.

3.7.3.1 Pseudo-Random Number Generator – PRNG-MGF-1

This PRNG is based on MDC-MGF-NTRU, defined in section 3.7.2.2 above. If it is called once only, its operation is indistinguishable from that of MGF-MDC-NTRU.

The function is parameterized by the following choice:

— A hash function *Hash* with output length *hLen* octets.

Input:
— An octet string *Z* of length *zLen* octets
— The desired length of the output, which is a positive integer *oLen*. (*oLen* shall be less than or equal to $hLen \times 2^{32}$).

Output: An octet string *o* of length *oLen* octets; or “error”.

Operation: The octet string *mask* shall be produced by the following or an equivalent sequence of steps:

```
1.  If zLen exceeds any input length limitation on the hash function Hash, output “error” and exit
2.  Initialize totLen to 0. Intialize remLen to 0.
3.  Initialize the octet string buf to be a string of zero octets of length hLen.
4.  Initialize counter:= 0.
5.  On input oLen:
    a.  Set totLen:=totLen + oLen.
    b.  If totLen exceeds $hLen \times 2^{32}$, output “error” and exit.
    c.  If remLen < oLen
          i.  Let M be the trailing remLen octets in buf.
         ii.  Let tmpLen:=oLen – remLen.
        iii.  Let cThreshold = counter + ceil[tmpLen/hLen].
         iv.  While counter < cThreshold do
                1.  Convert counter to an octet string C of length 4 octets using I2OSP.
                2.  Compute Hash(Z || C) with the selected hash function to produce an
                    octet string H of length hLen octets.
                3.  Let M = M || H.
                4.  Increment counter by one. If tmpLen > hLen, decrement tmpLen by
                    hLen.
          v.  Set remLen:=hLen – tmpLen. Set buf:=H.
    d.  else
          i.  Set M equal to the trailing remLen octets of buf.
         ii.  Set remLen:=remLen – oLen.
6.  Output the leading oLen octets of M as the octet string o.
```

3.7.3.2 Pseudo-Random Number Generator – PRNG-MDC-NTRU

This PRNG is based on a variant of MGF1. It is designed for the case where the output length of the core hash function is short. The only hash function supported for use with this MGF is MDC-2DES-NTRU, specified in section 3.7.1.1 of this document.

The function is parameterized by the following choice:
— A hash function *Hash* with output length *hLen* octets.

Input:
— An octet string *Z* of length *zLen* octets
— The desired length of the output, which is a positive integer *oLen*. (*oLen* shall be less than or equal to $hLen \times 2^{64}$).

Output: An octet string *mask* of length *oLen* octets; or “error”.

Operation: The octet string *mask* shall be produced by the following or an equivalent sequence of steps:

```
1.  If zLen exceeds any input length limitation on the hash function Hash, output “error” and exit
2.  Initialize totLen to 0. Intialize remLen to 0.
```

```
3.  Initialize the octet string buf to be a string of zero octets of length hLen.
4.  Initialize counter:= 0.
5.  On input oLen:
    e. Set totLen:=totLen + oLen.
    f. If totLen exceeds $hLen \times 2^{64}$, output “error” and exit.
    g. If $remLen < oLen$
         i. Let M be the trailing remLen octets in buf.
        ii. Let $tmpLen := oLen – remLen$.
       iii. Let $cThreshold = counter + \text{ceil}[tmpLen/hLen]$.
        iv. While $counter < cThreshold$ do
              1. Convert counter to an octet string C of length 8 octets using I2OSP.
              2. Compute $Hash(C || Z)$ with the selected hash function to produce an
                 octet string H of length hLen octets.
              3. Let $M = M || H$.
              4. Increment counter by one. If $tmpLen > hLen$, decrement tmpLen by
                 hLen.
         v. Set $remLen := hLen – tmpLen$. Set $buf := H$.
    h. else
         i. Set M equal to the trailing remLen octets of buf.
        ii. Set $remLen := remLen – oLen$.
6.  Output the leading oLen octets of M as the octet string o.
```

3.7.4 Random Number Generation

In various operations specified in this standard such as key generation and signature generation, the generation of random numbers is required. This standard strongly recommends the use of a secure random number generation method such as those methods that are approved by NIST in the FIPS series of standards (see [FIP00]) and by ANSI in the X9 series of standards (see [ANS98a] [ANS98b] [ANS98c]).

4 NTRUEncrypt Encryption Scheme (SVES)

The following section defines the EESS #1 supported encryption schemes. The only encryption scheme currently supported by EESS #1 is SVES. SVES stands for Shortest Vector Encryption Scheme (see [IEEE P1363.1] for more information).

4.1 NTRUEncrypt Encryption Scheme (SVES) Overview

The general NTRUEncrypt encryption scheme is a sequence of operations that are performed based on the choices of the NTRUEncrypt supporting algorithms, NTRUEncrypt primitives and the NTRUEncrypt parameters. In order to perform all of the NTRUEncrypt encryption scheme operations, the following NTRUEncrypt components must be specified:

NTRUEncrypt Domain Parameters –

1. Degree N
2. Small modulus p
3. Big modulus q

NTRUEncrypt Security Parameters –

1. Private key space D_f
2. Key generation primitive (KGP)
3. Temporary polynomial space D_g

NTRUEncrypt Scheme Options –

1. Random component size *db*
2. Message length encoding length *lLen, if required*
3. Message representative generation method (MRGM)
    a. Supported mask generation function (MGF)
    b. Hash function used to instantiate MGF (Hash)
4. Blinding value generation method (BVGM)
    a. Supported pseudo-random number generator (PRNG)
    b. Hash function used to instantiate PRNG (Hash – must be the same as the Hash used in the MRGM)
    c. Blinding value space D_r
    d. Random polynomial generation constant c
5. Decryption primitive (DP)

Note that since the public key space D_h is uniquely determined from D_f and D_g, it is not listed above as a required component for the NTRUEncrypt encryption scheme.

To illustrate the way that the NTRUEncrypt encryption scheme could be used, below is a step-by-step example of the processes that might occur when implementing the scheme. For simplicity, this example will use two entities, the encryptor Ernest and the decryptor Donna. The exact operations for key generation, encryption and decryption are spelled out in section 4.2.

1. Donna and Ernest agree on a set of domain parameters N, p, q. (these will be used throughout, but not referenced explicitly)
2. Donna and Ernest agree on the scheme options MGF, PRNG, Hash, db, D_r, c, BVGM, EP, PDP, DP.
3. Donna chooses her security parameters KGP, D_f, D_g.
4. Donna generates an NTRUEncrypt key pair (f, h) (using D_f, D_g and KGP).
5. Donna sends the public key h to Ernest (note this message need not be encrypted, but Ernest should have some assurance that it actually came from Donna).
6. Ernest chooses a message m to encrypt to Donna.
7. Ernest generates a random component b (using db and a random number generator) and then calculates the padded message pm and the seed *seed*. From *seed* he obtains the blinding value r (using PRNG, Hash, c and BVGM).Then from r and pm he obtains the message representative i (using MGF and Hash) and the ciphertext e.
8. Ernest sends the ciphertext e to Donna.
9. Donna uses DP with the private key f and the ciphertext e to find a message representative candidate i'.
10. Donna recovers the candidate padded message pm' and the candidate seed *seed*' (using the inverse of MPM) and then performs the encryption on m' (using *seed*', i', PRNG, Hash, c, BVGM and EP) to retrieve the expected encryption value e'.
11. If the resulting e' is the same as the received e and there were no padding errors, Donna knows that she decrypted the message properly and obtains the original message $m = m'$. If the resulting message e' is not the same as e or if there was a padding error, Donna outputs “fail”.

4.2 NTRUEncrypt Encryption Scheme (SVES) Operations

The NTRUEncrypt encryption scheme consists of the three operations – key generation, encryption and decryption. These three operations are defined generally in this section without assuming any specific choices of the NTRUEncrypt components listed in section 4.1.

4.2.1 NTRUEncrypt Key Generation

This section defines the NTRUEncrypt key generation operation. Note that within the definition of the NTRUEncrypt spaces may be definitions of additional variables (e.g. when defining D_f, the values $df1$, $df2$ and $df3$ may be specified as well as the appropriate method of combining them)

NTRUEncrypt Components:

— The NTRUEncrypt parameters N, q, p

— The NTRUEncrypt spaces D_f, D_g

— The selected NTRUEncrypt key generation primitive *KGP*

Input: None

Output: An NTRUEncrypt key pair (f, h)

Operation: The NTRUEncrypt key pair generation shall be computed by the following or an equivalent sequence of steps:

1. Using *KGP*, with inputs N, q, p, D_f, D_g, generate an NTRUEncrypt key pair (f, h)
2. Return (f, h)

4.2.2 NTRUEncrypt Encryption Scheme SVES-3: Encryption

This section defines the NTRUEncrypt encryption operation for SVES-3. Note that within the definition of the NTRUEncrypt spaces may be definitions of additional variables (e.g. when defining D_r, the values $dr1$, $dr2$ and $dr3$ may be specified as well as the appropriate method of combining them).

Components:
— The length of the encoded length *lLen*.
— The number of bits of random data *db*, which must be a multiple of 8.
— The chosen Mask Generation Function and Hash Function.
— The chosen Blinding Value Generation Method and the associated parameters
— The *OID*, an octet string
— The number of bits of public key to hash, *pkLen*.

Inputs:
— The message *m*, which is an octet string of length *l* octets
— The public key *h*

Output:
— The ciphertext *e*, which is a ring element, or "message too long"

Operation: The ciphertext *e* shall be calculated by the following or an equivalent sequence of steps:

1. Calculate:
   a. *nLen* = ceil [*N*/8], the number of octets required to hold *N* bits.
   b. *octL* = the *lLen*-octet-long encoding of the message length *l*.
   c. *bLen* = *db*/8, the length in octets of the random data.
   d. *maxLen* = *nLen* - 1 - *lLen* - *bLen*, the maximum message length.
2. If *l* > *maxLen*, output "message too long" and stop.
3. Randomly select an octet string *b* of length *bLen*.
4. Form the octet string *p0*, consisting of the 0 byte repeated (*maxLen* + 1 - *l*) times.
5. Form the octet string *M* of length *nLen* as
   b || *octL* || *m* || *p0*.
6. Form the octet string *hTrunc*, consisting of the first *pkLen* bits of the packed representation of the public key *h* (generated using RE2POSP, section 2.3.6). Form *sData* as the octet string
   OID || *m* || *b* || *hTrunc*
7. Use the chosen blinding value generation method with the seed *sData* and the chosen parameters to produce *r*.
8. Calculate $R = r*h \bmod q$.
9. Calculate $R2 = R \bmod 2$.
10. Convert *R2* to the octet string *oR2* using BE2OSP.
11. Form *m*' by putting *oR2* through the chosen MGF/Hash and XORing the leading *nLen* bytes of the output with *M*.
12. Set the leading ((*nLen* * 8) - *N*) bits of the final octet of *m*' to 0.
13. If
    $(N-q)/2 < m'(1) < (N+q)/2$,
    return to step 3.
14. Convert *m*' to *i*, a binary polynomial of length *N*, using OS2BEP.

15. Calculate the ciphertext as $e = R + i \bmod q$.

4.2.3 NTRUEncrypt Encryption Scheme SVES-3: Decryption

This section defines the NTRUEncrypt decryption operation for SVES-3. Note that within the definition of the NTRUEncrypt spaces may be definitions of additional variables (e.g. when defining D_r, the values $dr1$, $dr2$ and $dr3$ may be specified as well as the appropriate method of combining them).

Components:

— The NTRUEncrypt decryption primitive to use
— The length of the encoded length *lLen*.
— The number of bits of random data *db*, which must be a multiple of 8.
— The chosen Mask Generation Function and Hash Function.
— The chosen Blinding Value Generation Method and the associated parameters
— The *OID*, an octet string
— The number of bits of public key to hash, *pkLen*.

Inputs:

— The ciphertext *e*, which is a polynomial of degree N-1.
— The private key *f* or (f, f_p).
— The public key *h*

Output:

— The message *m*, which is an octet string, or "fail".

Operation: The message *m* shall be calculated by the following or an equivalent sequence of steps:

1. Calculate:
    a. *nLen* = ceil [*N*/8], the number of octets required to hold *N* bits.
    b. *bLen* = *db*/8, the length in octets of the random data
    c. *maxLen* = *nLen* - 1 - *lLen* - *bLen*, the maximum message length.
2. Decrypt the ciphertext *e* using the selected NTRU decryption primitive with inputs *e* and *f* to get the candidate decrypted polynomial *ci*.
3. Calculate the candidate value for $r*h$, $cR = e - ci$.
4. Calculate $cR2 = cR \bmod 2$.
5. Convert *cR2* to the octet string *coR2* using BE2OSP.
6. Convert the binary polynomial *ci* to the octet string *cm*' using BE2OSP.
7. Form *cm* by putting *coR2* through the chosen MGF/Hash and XORing the leading *nLen* bytes of the output with *cm*'.
8. Set the leading ((*nLen* * 8) - N) bits of the final octet of *cM* to 0.
9. Parse *cM* as follows.
    a. The first *bLen* octets are the octet string *cb*.
    b. The next *lLen* octets represent the message length. Convert the value stored in these octets to the candidate message length *cl*. If *cl* > *maxLen*, set *fail* = 1 and set *cl* = *maxL*.
    c. The next *cl* octets are the candidate message *cm*. the remaining octets should be 0. If they are not, set fail = 1.
10. Form the octet string *hTrunc*, consisting of the first *pkLen* bits of the packed representation of the public key *h* (generated using RE2POSP, section 2.3.6). Form *sData* as the octet string
    OID || *cm* || *cb* || *hTrunc*
11. Use the chosen blinding value generation method with the seed *sData* and the chosen parameters to produce *r*.
12. Calculate $cR' = h * cr \bmod q$.
13. If *cR*' != *cR*, set *fail* = 1

14. If *fail* = 1, output "fail". Otherwise, output *cm* as the decrypted message *m*.

4.3 Supported Parameter Sets

This section defines specific sets of parameters for the NTRUEncrypt encryption scheme (SVES) that are supported by the EESS #1 standard. The parameters chosen in these sets must be used as a group and may not be mixed and matched. Each parameter set is chosen to maximize both security and efficiency for the selected security level. The best known attacks on these parameter sets appear to be measurable based on the parameter N, i.e. a larger N represents a stronger security level. No other sets of parameters shall be used.

The parameter sets ees139ep1, ees139ep2, ees251ep1, ees251ep2, ees251ep3, ees347ep1, ees503ep1 that appeared in previous versions of this document are deprecated and should not be used.

4.3.1 ees251ep4

This section specifies the selected domain parameters, security parameters, primitives, encoding methods and supporting methods for the parameter set **ees251ep4**. This parameter set name stands for efficient embedded security (ees) encryption parameters (ep) with degree 251 (251) set 4 (4). The object identifier for this parameter set is specified in section A.1.

$N = 251$
$p = 2$
$q = 239$

4.3.1.1 Key generation:

KGP-NTRU2 with
    $dF = 72$
    $D_g = g$, where $dg = 72$

4.3.1.2 Encryption/Decryption:

SVES-3 encryption and decryption as in sections 4.2.2 and 4.2.3, parameterized as follows:

$lLen = 1$
$db = 80$
SVDP-NTRU2
MGF-1 with
    SHA-1 (MGF)
BVGM-NTRU1 with
    PRNG-MGF-1 with *SHA-1* (PRNG)
    $dr = 72$
    $c = 8$

OID = 00 01 04
pkLen = 80

4.3.2 ees251ep5

This section specifies the selected domain parameters, security parameters, primitives, encoding methods and supporting methods for the parameter set **ees251ep5**. This parameter set name stands for efficient embedded security (ees) encryption parameters (ep) with degree 251 (251) set 4 (5). The object identifier for this parameter set is specified in section A.1.

$N = 251$
$p = 2$
$q = 239$

4.3.2.1 Key generation:

KGP-NTRU2 with
    dF = 72
    $D_g = g$, where *dg* = 72

4.3.2.2 Encryption/Decryption:

SVES-3 encryption and decryption as in sections 4.2.2 and 4.2.3, parameterized as follows:

lLen = 1
db = 80
SVDP-NTRU2
MGF-1 with
    MDC-NTRU (MGF)
BVGM-NTRU1 with
    PRNG-MGF-1 with *MDC-NTRU* (PRNG)
    dr = 72
    $c = 8$
OID = 00 01 05
pkLen = 80

5 NTRUSign Signature Scheme (SVSSA)

The following section defines the EESS #1 supported signature schemes. The only signature scheme currently supported by EESS #1 is SVSSA. SVSSA stands for Shortest Vector Signature Scheme with Appendix (see [IEEE P1363.1] for more information).

5.1 NTRUSign Signature Scheme (SVSSA) Overview

The general NTRUSign signature scheme is a sequence of operations that are performed based on the choices of the NTRUSign supporting algorithms, NTRUSign primitives and the NTRUSign parameters. In order to perform all of the NTRUSign signature scheme operations, the following NTRUSign components must be specified.

NTRUSign Domain Parameters –

1. Degree N
2. Big modulus q

NTRUSign Security Parameters –

1. Private key spaces D_f, D_g
2. Basis completion space D_{FG}
3. Basis completion maximum adjustment *MaxAdjustment* (optional)
4. Signature failure tolerance *SignFailTolerance*
5. Key norm bound *KeyNormBound* (optional)
6. Key generation primitive (KGP)

NTRUSign Scheme Options –

1. Signature norm bound *NormBound*
2. Message randomization element space D_r
3. Message representative generation method (MRGM)
    a. Supported hash function (Hash)
    b. Supported pseudo-random number generator (PRNG)
    c. Supported core hash function for the PRNG (PrngHash)
    d. Random polynomial generation constant c (optional)
4. Signature primitive (SP)
5. Verification primitive (VP)

Note that since the public key space D_h is uniquely determined from D_f and D_g, it is not listed above as a required component for the NTRUSign signature scheme.

To illustrate the way that the NTRUSign signature scheme could be used, below is a step-by-step example of the processes that might occur when implementing the scheme. For simplicity, this example will use two entities, the signer Samantha and the verifier Victor. The exact operations for key generation, signing and verifying are spelled out in section 5.2.

1. Samantha and Victor agree on a set of domain parameters *N*, *q*. (these will be used throughout, but not referenced explicitly)
2. Samantha and Victor agree on the scheme options Hash, PRNG, PrngHash, *c*, *NormBound*, D_r, MRGM, SP, VP.
3. Samantha chooses her security parameters KGP, D_f, D_g, D_{FG}, *KeyNormBound*, *MaxAdjustment*, *SignFailTolerance*.
4. Samantha generates an NTRUSign key pair ((*f*, *g*), *h*) and basis completion pair (*F*, *G*) (using KGP, D_f, D_g, D_{FG}, *KeyNormBound* and *MaxAdjustment*)
5. Samantha send the public key *h* to Victor (note this message need not be encrypted, but Victor should have some assurance that it actually came from Samantha).
6. Samantha chooses a message *m* to sign for Victor.
7. Samantha selects a message randomization value *r*, computes the message representative *i* and generates a signature *s* on *m* (using SP, D_r, MRGM, Hash, PRNG, *c*, *SignFailTolerance*, *f*, and *F*).
8. Samantha may optionally check the signature (using VP, *NormBound*, *r*, *s* and *i*)
9. Samantha sends the message *m*, the message randomization value *r* and the signature *s* to Victor (again these can be sent in the clear and Victor does not have to know ahead of time that it came from Samantha)
10. Victor checks the signature on *m* (using MRGM, Hash, PRNG, *c*, VP, *NormBound*, *r* and *s*).
11. If the signature passes, Victor trusts that Samantha generated the signature on *m*.

5.2 NTRUSign Signature Scheme (SVSSA) Operations

The NTRUSign signature scheme consists of three operations – key generation, signature generation and verification.

5.2.1 NTRUSign Key Generation

This section defines the NTRUSign key generation operation. Note that within the definition of the NTRUSign spaces may be definitions of additional variables, although none of the parameter sets specified in this standard use these additional variables.

NTRUSign Components:

— The NTRUSign domain parameters *N*, *q*,

— The NTRUSign security parameters *KeyNormBound* (optional), D_f, D_g, D_{FG}, *MaxAdjustment* (optional), *perturbationBases*.

— The NTRUSign *basisType* variable, equal to "standard" or "transpose".

— The selected NTRUSign key generation primitive *KGP*

Input: None

Output: An NTRU key pair consisting of the private key (*f*, *g*, *D*) and the public key *h*

Operation: The NTRU key pair shall be computed by the following or an equivalent sequence of steps:

1. Using *KGP*, with inputs *N*, *q*, D_f, D_g, D_{FG}, *perturbationBases*, *basisType*, *KeyNormBound* (optional), *MaxAdjustment* (optional), generate an NTRUSign private key (f_i, f'_i, h_i) for $0 \le i \le$ *perturbationBases* and public key *h*.

5.2.2 NTRUSign Signature Operation

This section defines the NTRUSign signature operation. Note that within the definition of the NTRUSign spaces may be definitions of additional variables (e.g. when defining D_i, the value *di* may be specified). Note that *h* need not be known if the verification primitive *VP* is performed using alternate methods using the private key (*f*, *g*).

NTRUSign Components:
- The NTRUSign parameters *N*, *q*
- The NTRUSign security parameter *NormBound*
- The NTRUSign message randomization space D_r
- The selected NTRUSign signature primitive *SP*
- (optional) The selected NTRUSign verification primitive *VP*
- (optional) The parameter *SignFailTolerance*
- The selected NTRUSign message representative generation method *MRGM*
- The selected Hash function *Hash*
- The selected pseudo-random number generation function *PRNG*
- The selected core hash function for the pseudo-random number generator *PrngHash*

Input:
- The signer's NTRUSign private key (f_i, f'_i, h_i) for $0 \le i \le perturbationBases$
- The message *m* to be signed, which is a octet string
- (optional) The signer's NTRUSign public key *h*

Output: The signature, which is a polynomial *s* and the message randomization value *r*

Operation: The signature *s* shall be computed by the following or an equivalent sequence of steps:

```
1.  Set integer SignFail := 0
2.  Select a message randomization value r from the space $D_r$ (this may be done randomly or
    deterministically)
3.  Using MRGM, with inputs m, r and components N, q, lr, Hash, PRNG, PrngHash, generate the
    message representative i.
4.  Using SP, with inputs f, F, i and components N, q, generate the signature s
5.  (optional) If verification checking is desired
      a. Using VP, with inputs s, h, i and components N, q, NormBound, verify the signature s
      b. If s is not a valid signature
           i.   Go to 2 (note that if the same r is chosen, the signature will always fail)
           ii.  Set SignFail := SignFail + 1
           iii. If SignFail > SignFailTolerance
                  1. Output "Signature Failed"
6.  Output s and r
```

5.2.3 NTRUSign Verification Operation

This section defines the NTRUSign verification operation.

NTRUSign Components:
- The NTRUSign parameters *N*, *q*
- The NTRUSign security parameter *NormBound*
- The NTRUSign message randomization space D_r
- The selected NTRUSign verification primitive *VP*
- The selected NTRUSign message representative generation method *MRGM*
- The selected hash function *Hash*
- The selected pseudo-random number generation function *PRNG*
- The selected core hash function for the pseudo-random number generator *PrngHash*

Input:
— The signer's NTRUSign public key *h*
— The signature to be verified, which is a polynomial *s*
— The message *m* for which *s* is alleged to be a signature
— The message randomization value *r*

Output: A message indicating that the signature is either "valid" or "invalid"

Operation: A signature *s* shall be verified by the following or an equivalent sequence of steps:

1. Using *MRGM*, with inputs *m*, *r* and components *N*, *q*, *lr*, *Hash*, *PRNG*, *PrngHash*, generate the message representative *i*.
2. Using *VP*, with inputs *s*, *h*, *i* and components *N*, *q*, *NormBound*, verify the signature *s*.
3. If the verification is successful, output "valid" and stop.
4. Output "invalid."

5.3 Supported Parameter Choices

This section defines specific sets of parameters for the NTRUSign signature scheme (SVSSA) that are supported by the EESS #1 standard. The parameters chosen in these sets must be used as a group and may not be mixed and matched. Each parameter set is chosen to maximize both security and efficiency for the selected security level. The best known attacks on these parameter sets appear to be measurable based on the parameter *N*, i.e. a larger *N* represents a stronger security level. No other sets of parameters shall be used.

The domain parameters for each parameter sets are required and shall be used whenever the parameter set is specified. The security parameters are recommended for use with the specified domain parameters. The primitives and encoding methods are typically specified at a higher level, but recommended choices are listed.

5.3.1 ees251sp2

This section specifies the selected domain parameters, security parameters, primitives, encoding methods and supporting methods for the parameter set **ees251sp2**. This parameter set name stands for efficient embedded security (ees) signature parameters (sp) with degree 251 (251) set 2 (2). The object identifier for this parameter set is specified in section A.1.

5.3.1.1 NTRUSign Domain Parameters (required)

$N = 251$
$q = 128$

5.3.1.2 NTRUSign Security Parameters (recommended)

$D_f = f$ where
    $df = 73$
$D_g = g$, where
    $dg = 71$
D_{FG} = any valid F, G

MaxAdjustment = 200
SignFailTolerance = 0
KeyNormBound = None
KGP-NTRUSign1
perturbationBases = 0
basisType = "standard"

5.3.1.3 NTRUSign Scheme Options (required)

NormBound = 310
D_r = random *r* where
    lr = 1 (e.g. *r* is a single random octet)
SVSP-NTRU
SVVP-NTRU
MRGM-NTRUSign2 with
    c = 8
    NumGroups = 13
    NumElements = 3
    SHA-1 (Hash)
    PRNG-MGF1 with *SHA-1* (PRNG)

5.3.2 ees251sp3

This section specifies the selected domain parameters, security parameters, primitives, encoding methods and supporting methods for the parameter set **ees251sp3**. This parameter set name stands for efficient embedded security (ees) signature parameters (sp) with degree 251 (251) set 3 (3). The object identifier for this parameter set is specified in section A.1.

5.3.2.1 NTRUSign Domain Parameters (required)

$N = 251$
$q = 128$

5.3.2.2 NTRUSign Security Parameters (recommended)

$D_f = f$ where
    $df = 73$
$D_g = g$, where
    $dg = 71$
D_{FG} = any valid *F*, *G*
MaxAdjustment = 200
SignFailTolerance = 0
KeyNormBound = None
KGP-NTRUSign1
perturbationBases = 0
basisType = "standard"

5.3.2.3 NTRUSign Scheme Options (required)

NormBound = 310

D_r = random *r* where

 lr = 1 (e.g. *r* is a single random octet)

SVSP-NTRU

SVVP-NTRU

MRGM-NTRUSign2 with

 c = 8

 NumGroups = 13

 NumElements = 3

 SHA-1 (Hash)

 PRNG-MGF1 with *MDC-NTRU* (PRNG)

5.3.3 ees251sp4

This section specifies the selected domain parameters, security parameters, primitives, encoding methods and supporting methods for the parameter set **ees251sp4**. This parameter set name stands for efficient embedded security (ees) signature parameters (sp) with degree 251 (251) set 4 (4). The object identifier for this parameter set is specified in section A.1.

5.3.3.1 NTRUSign Domain Parameters (required)

N = 251

q = 128

5.3.3.2 NTRUSign Security Parameters (recommended)

D_f = *f* where

 df = 73

D_g = *g*, where

 dg = 71

D_{FG} = any valid *F*, *G*

MaxAdjustment = 200

SignFailTolerance = 0

KeyNormBound = None

KGP-NTRUSign1

perturbationBases = 0

basisType = “standard”

5.3.3.3 NTRUSign Scheme Options (required)

NormBound = 310

D_r = random *r* where

 lr = 1 (e.g. *r* is a single random octet)

SVSP-NTRU

SVVP-NTRU

MRGM-NTRUSign1 with

 SHA-1 (Hash)

 PRNG-MGF1 with *SHA-1* (PRNG)

5.3.4 ees251sp5

This section specifies the selected domain parameters, security parameters, primitives, encoding methods and supporting methods for the parameter set **ees251sp5**. This parameter set name stands for efficient embedded security (ees) signature parameters (sp) with degree 251 (251) set 5 (5). The object identifier for this parameter set is specified in section A.1.

5.3.4.1 NTRUSign Domain Parameters (required)

N = 251
q = 128

5.3.4.2 NTRUSign Security Parameters (recommended)

D_f = f where
    df = 73
D_g = g, where
    dg = 71
D_{FG} = any valid F, G
MaxAdjustment = 200
SignFailTolerance = 0
KeyNormBound = None
KGP-NTRUSign1
perturbationBases = 0
basisType = "standard"

5.3.4.3 NTRUSign Scheme Options (required)

NormBound = 310
D_r = random r where
    lr = 1 (e.g. r is a single random octet)
SVSP-NTRU
SVVP-NTRU
MRGM-NTRUSign1 with
    SHA-1 (Hash)
    PRNG-MGF1 with *MDC-NTRU* (PRNG)

5.3.5 ees251sp6

This section specifies the selected domain parameters, security parameters, primitives, encoding methods and supporting methods for the parameter set **ees251sp6**. This parameter set name stands for efficient embedded security (ees) signature parameters (sp) with degree 251 (251) set 6 (6). The object identifier for this parameter set is specified in section A.1.

5.3.5.1 NTRUSign Domain Parameters (required)

N = 251
q = 128

5.3.5.2 NTRUSign Security Parameters (recommended)

D_f = *f* where
    df = 73
D_g = *g*, where
    dg = 71
D_{FG} = any valid *F*, *G*
MaxAdjustment = 200
SignFailTolerance = 0
KeyNormBound = None
KGP-NTRUSign1
perturbationBases = 1
basisType = "transpose"

5.3.5.3 NTRUSign Scheme Options (required)

NormBound = 310
D_r = random *r* where
    lr = 1 (e.g. *r* is a single random octet)
SVSP-NTRU
SVVP-NTRU
MRGM-NTRUSign2 with
    c = 8
    NumGroups = 13
    NumElements = 3
    SHA-1 (Hash)
    PRNG-MGF1 with *SHA-1* (PRNG)

5.3.6 ees251sp7

This section specifies the selected domain parameters, security parameters, primitives, encoding methods and supporting methods for the parameter set **ees251sp7**. This parameter set name stands for efficient embedded security (ees) signature parameters (sp) with degree 251 (251) set 3 (7). The object identifier for this parameter set is specified in section A.1.

5.3.6.1 NTRUSign Domain Parameters (required)

N = 251
q = 128

5.3.6.2 NTRUSign Security Parameters (recommended)

D_f = *f* where
    df = 73
D_g = *g*, where
    dg = 71
D_{FG} = any valid *F*, *G*
MaxAdjustment = 200
SignFailTolerance = 0
KeyNormBound = None

KGP-NTRUSign1
perturbationBases = 1
basisType = “transpose”

5.3.6.3 NTRUSign Scheme Options (required)

NormBound = 310
D_r = random *r* where
    lr = 1 (e.g. *r* is a single random octet)
SVSP-NTRU
SVVP-NTRU
MRGM-NTRUSign2 with
    c = 8
    NumGroups = 13
    NumElements = 3
    SHA-1 (Hash)
    PRNG-MGF1 with *MDC-NTRU* (PRNG)

5.3.7 ees251sp8

This section specifies the selected domain parameters, security parameters, primitives, encoding methods and supporting methods for the parameter set **ees251sp8**. This parameter set name stands for efficient embedded security (ees) signature parameters (sp) with degree 251 (251) set 8 (8). The object identifier for this parameter set is specified in section A.1.

5.3.7.1 NTRUSign Domain Parameters (required)

N = 251
q = 128

5.3.7.2 NTRUSign Security Parameters (recommended)

D_f = *f* where
    df = 73
D_g = *g*, where
    dg = 71
D_{FG} = any valid *F*, *G*
MaxAdjustment = 200
SignFailTolerance = 0
KeyNormBound = None
KGP-NTRUSign1
perturbationBases = 1
basisType = “transpose”

5.3.7.3 NTRUSign Scheme Options (required)

NormBound = 310
D_r = random *r* where
    lr = 1 (e.g. *r* is a single random octet)
SVSP-NTRU

SVVP-NTRU
MRGM-NTRUSign1 with
    SHA-1 (Hash)
    PRNG-MGF1 with *SHA-1* (PRNG)

5.3.8 ees251sp9

This section specifies the selected domain parameters, security parameters, primitives, encoding methods and supporting methods for the parameter set **ees251sp9**. This parameter set name stands for efficient embedded security (ees) signature parameters (sp) with degree 251 (251) set 9 (9). The object identifier for this parameter set is specified in section A.1.

5.3.8.1 NTRUSign Domain Parameters (required)

$N = 251$
$q = 128$

5.3.8.2 NTRUSign Security Parameters (recommended)

$D_f = f$ where
    $df = 73$
$D_g = g$, where
    $dg = 71$
D_{FG} = any valid F, G
MaxAdjustment = 200
SignFailTolerance = 0
KeyNormBound = None
KGP-NTRUSign1
perturbationBases = 1
basisType = “transpose”

5.3.8.3 NTRUSign Scheme Options (required)

NormBound = 310
D_r = random r where
    $lr = 1$ (e.g. r is a single random octet)
SVSP-NTRU
SVVP-NTRU
MRGM-NTRUSign1 with
    SHA-1 (Hash)
    PRNG-MGF1 with *MDC-NTRU* (PRNG)

6 ASN.1 Syntax

This section covers the representation of cryptographic objects used in NTRUEncrypt in terms of ASN.1 Syntax. This is important for use with certificates, certificate revocation and other cryptographic messages. In particular, ASN.1 syntax is used to represent the contents of X.509 certificates. Some additional object identifiers and placeholders for ASN.1 syntax for NTRUSign are included in the ASN.1 module in Annex A for informational purposes.

6.1 General Types

6.1.1 General Vector Types

This section defines the ASN.1 syntax for vector types that are used to represent polynomials for NTRUEncrypt. There are three primary types of vectors – public vectors, binary vectors and trinary vectors. Public vectors are polynomials that have N coefficients that are reduced modulo q. Binary vectors are polynomials that have N coefficients that are reduced modulo p, where $p = 2 + X$, making the coefficients all either 0 or 1. Trinary vectors are polynomials that have N coefficients that are reduced modulo p, where $p = 3$, making the coefficients either 0, 1 or –1. Coefficients in the vectors are always represented as positive integers, however since the coefficients are taken modulo either p or q, they should be reduced into the appropriate interval before being used (e.g. modulo 3 numbers are reduced to 0, 1 or -1 and modulo q numbers are usually reduced into the interval $–q/2 <= x <= q/2$).

All of the vector types consist of a string of integer values that are concatenated and stored in an OCTET STRING. Each integer is encoded by taking the smallest positive representation of the integer modulo p, q or N (e.g. taking –1 as a mod 3 number gives you the integer 2), encoding it with I2OSP, and then using OS2BSP (see section 2.3) to obtain a bit string of the appropriate length (e.g. truncating the leftmost 6 bits to obtain 2 bits for a mod 3 coefficient). The integer is recovered by obtaining the correct bit string (e.g. for packed NTRUEncrypt 251, each coefficient is represented by 7 bits), and using BS2OSP and then OS2IP (see section 2.3). So, to encode the value 55 as a 7-bit value, the integer is encoded as the octet 00110111 (using I2OSP) and then it is truncated on the left (using OS2BSP with the desired length set to 7) to obtain the bit string 0110111. To obtain the value of the coefficient represented by the bit string 0110111, the bit string is expanded on the left to obtain the octet 00110111 (using BS2OSP) and then converted to the integer 55 (using OS2IP).

```
NTRUPublicVector ::= CHOICE {
        modQVector            [0] IMPLICIT ModQVector,
        packedModQVector      [1] IMPLICIT PackedModQVector,
        …
}

NTRUBinaryVector ::= CHOICE {
        listedBinaryVector    [0] IMPLICIT ListedBinaryVector,
        packedBinaryVector    [1] IMPLICIT PackedBinaryVector,
        modQVector            [2] IMPLICIT ModQVector,
        …
}
```

Binary vectors in NTRUEncrypt usually arise from reducing some polynomial mod 2 + X. When a polynomial is reduced mod 2 + X using the techniques given in EESS#1, it is possible to get an “exception case result”, which is the polynomial $2 + X^2 + X^4 + \ldots + X^{N-1}$. This exception case cannot be encoded as a listed or packed binary vector, and must be encoded as a **ModQVector**.

ModQVector ::= OCTET STRING

The contents of this **OCTET STRING** are obtained by converting a ring element to a octet string using the RE2OSP conversion primitive. A **ModQVector** is a representation of a polynomial of degree N and must include all N coefficients, even if the high-order ones are zero.

This is the preferred format for public keys, encrypted values and signatures.

PackedModQVector ::= OCTET STRING

The contents of this **OCTET STRING** are obtained by converting a ring element to a octet string using the RE2POSP conversion primitive. A **PackedModQVector** is a representation of a polynomial of degree N and must include all N coefficients, even if the high-order ones are zero.

For the NTRUEncrypt parameter sets in this standard, there is no difference between the **ModQVector** representation and the **PackedModQVector** representation. For the NTRUSign parameter sets, use of the **PackedModQVector** representation may result in saving about N bits per vector.

ListedBinaryVector ::= OCTET STRING

This **OCTET STRING** is to be interpreted as a sequence of 1-byte (for $N < 256$) or 2-byte (for $256 \leq N < 512$) unsigned integers. Each integer corresponds to a coefficient of the polynomial that is equal to 1. All coefficients that are not included in the list are equal to 0. As an example, the byte 0x25 (for $N = 251$) would indicate that the coefficient of the polynomial of degree 37 (i.e. X^{37}) is 1. The integers must be listed in ascending order numerically and no coefficient may be listed more than once. This is the preferred format for NTRUEncrypt private key components.

PackedBinaryVector ::= OCTET STRING

This **OCTET STRING** is to be interpreted as a sequence of 1-bit unsigned integers. These integers are packed into the **OCTET STRING** starting from the least significant bit of the first byte, without any additional padding, irrespective of the byte boundaries of the **OCTET STRING**. The most significant bits of the final byte of the **OCTET STRING** are padded with 0's if necessary. Each integer corresponds to a single coefficient value x in the range $0 <= x <= 1$, ordered from lowest degree to highest. For NTRUEncrypt-251 (or NTRUSign-251), NTRUEncrypt-347 and NTRUEncrypt-503, a **PackedBinaryVector** will take up 32 bytes (the last 5 bits are set to 0), 44 bytes (the last 5 bits are set to 0) and 63 bytes (the last bit is set to 0) respectively.

Note that the contents of a **PackedBinaryVector** are **different** from the result of encoding the binary vector with BRE2OSP.

We define two further types, which can be used to represent a polynomial of arbitrary degree:

```
NTRUGeneralPolynomial ::= SEQUENCE {
        numberOfEntries      INTEGER,
        modulus              INTEGER,
        coefficients         GeneralVector
}
```

This **SEQUENCE** defines a polynomial with any integer modulus and any degree. The **INTEGER** called **numberOfEntries** is equal to the degree + 1 of the polynomial and represents the number of coefficients to be listed. The **INTEGER** called **modulus** is a modulus or more generally, **modulus** is an upper bound on the value of the coefficients of the polynomial. The **GeneralVector** called **coefficients** is the concatenation of the values of the coefficients of the polynomial, obtained by treating the polynomial as if it were of degree **numberOfEntries**–1, converting this polynomial to an octet string using RE2OSP, and encoding the result as a **ModQVector**.

```
GeneralVector ::= OCTET STRING
```

This OCTET STRING is to be interpreted as a **ModQVector** except that only coefficients of the polynomial up to the specified **numberOfEntries** are included in the **OCTET STRING**. So, for NTRUEncrypt-251 with a modulus q of 128, the polynomial $2 + X$ (which is the small modulus p) could be encoded as an **NTRUGeneralPolynomial** with **numberOfEntries** equal to 2, **modulus** equal to 128, and a **GeneralVector** whose value is two bytes long.

6.1.2 Object Identifiers

This standard uses the following base object identifiers.

```
ntru OBJECT IDENTIFIER ::= {
        iso(1) identified-organization(3) dod(6) internet(1) private(4) enterprises(1)
        ntruCryptosystems (8342) }

        id-eess1 OBJECT IDENTIFIER ::= { ntru eess (1) eess-1 (1) }

id-eess1-algs OBJECT IDENTIFIER ::= {id-eess1  1}
id-eess1-params OBJECT IDENTIFIER ::= {id-eess1  2}
id-eess1-encodingMethods OBJECT IDENTIFIER ::= {id-eess1 3}
```

6.2 ASN.1 for NTRUEncrypt SVES

This section defines the ASN.1 object identifiers for NTRUEncrypt keys and NTRUEncrypt encrypted data, and defines the types **NTRUPublicKey**, **NTRUPrivateKey**, **NTRUEncryptedData**, and **EESS1v1-SVES-Parameters**.

The object identifier **id-ntru-EESS1v1-SVES** identifies NTRUEncrypt public and private keys and NTRUEncrypt-encrypted data. When this object identifier is used in an **AlgorithmIdentifier**, the parameters shall be of type **EESS1v1-SVES-Parameters**.

Note that EESS#1 breaks with common practice in requiring that a key be encoded with the scheme parameters (such as a mask generation function identifier for NTRUEncrypt or the verification bounds for NTRUSign) as well as with the algorithm domain parameters (such as *N*, *q* and *p*). Ensuring that a key can only be used in one scheme provides a defense against version rollback attacks and is good security practice.

This section of this standard only defines ASN.1 for the currently supported parameter sets. ASN.1 for previously parameter sets will appear in a future appendix to this standard.

6.2.1 NTRUEncrypt Public Keys

NTRUEncrypt public keys are identified by the following object identifier:

```
id-ntru-EESS1v1-SVES OBJECT IDENTIFIER  ::= {id-eess1-algs  1}
```

The parameters field associated with this OID in an **AlgorithmIdentifier** shall have the type **EESS1v1-SVES-params**, defined in section 6.2.4 below.

NTRUEncrypt public keys should be represented with the following syntax:

```
NTRUPublicKey ::= SEQUENCE {
        publicKeyVector        NTRUPublicVector,  -- h
        ntruKeyExtensions      NTRUKeyExtensions OPTIONAL
        }

NTRUKeyExtensions ::= SEQUENCE SIZE(1..MAX) OF NTRUKeyExtension

NTRUKeyExtension ::= CHOICE {
        keyID             [0] IMPLICIT INTEGER,
        …}
```

The fields of the type **NTRUPublicKey** have the following meanings:

- **publicKeyVector** is the polynomial h. If the **NTRUPublicVector** is a **ModQVector**, each coefficient will be represented by one byte starting with the lowest degree and going to the highest. If the **NTRUPublicVector** is a **PackedModQVector**, this is the octet string representing h obtained using RE2BSP and then BS2OSP. All coefficients up to X^{N-1} shall be explicitly included in **publicKeyVector**. Representing the NTRUEncrypt public key as a **ModQVector** is the preferred method.
- **ntruKeyExtensions** is provided for future extensibility. Only one extension is defined in EESS#1.

The fields of the type **NTRUKeyExtension** have the following meanings:

- **keyID** can be used to associate a unique key identifier with the key.

- The "…" is used to indicate this object is extensible. Additional types may be added in future versions.

6.2.2 NTRUEncrypt Private Keys

NTRUEncrypt private keys are identified by the following object identifier:

```
id-ntru-EESS1v1-SVES OBJECT IDENTIFIER  ::= {id-eess1-algs  1}
```

They are distinguished from NTRUEncrypt public keys by form and by context. The parameters field associated with this OID in an **AlgorithmIdentifier** shall have the type **EESS1v1-SVES-params**, defined in section 6.2.4 below.

An NTRUEncrypt private key should be represented with the following syntax:

```
NTRUPrivateKey ::= SEQUENCE {
        version                 INTEGER,
        publicKeyVector         NTRUPublicVector OPTIONAL,
        privateKeyType          PrivateKeyType,
        ntruPrivateKeyVectors NTRUPrivateKeyVectors,
        …}

PrivateKeyType ::= INTEGER

NTRUPrivateKeyVectors ::= SEQUENCE {
        fVectors        FVectors,
        gVectors        GVectors OPTIONAL }

FVectors ::= SEQUENCE OF NTRUBinaryVector

GVectors ::= SEQUENCE OF NTRUBinaryVector
```

The fields of the type **NTRUPrivateKey** have the following meanings:

- **version** is the version number, for compatibility with future revisions of this document. It shall be 0 for this version of the document.
- **publicKeyVector** is the public key associated with the private key. To complete the ciphertext validity check when decrypting, the decrypter must know the public key. It can be provided either explicitly in this field, or implicitly by providing the **GVectors** in the **ntruPrivateKeyVectors** field.
- **privateKeyType** determines the format of the private key vector. Type 1 keys have the form f = 1 + *p**(f1*f2 + f3) with f1, f2 and f3 listed in that order in the **ntruPrivateKeyVectors** field. Type 2 keys have the form f = 1 + *p**F with F listed in the **ntruPrivateKeyVectors** field.

The fields of the type **NTRUPrivateKeyVectors** have the following meanings:

- **fVectors** contains the f-vectors: f1, f2 and f3 if the key is of type1, F if the key is of type 2. The preferred format for each **FVector** is **ListedBinaryVector**.

- **gVectors** contains the vector g. This field need only be included if the **publicKeyVector** field in the NTRUPRIVATEKEY is omitted. As with **FVectors**, the preferred format for each **GVector** is **ListedBinaryVector**.

6.2.3 NTRUEncrypt Encrypted Data

NTRUEncrypt encrypted data are identified by the following object identifier:

```
id-ntru-EESS1v1-SVES OBJECT IDENTIFIER  ::= {id-eess1-algs 1}
```

The parameters field associated with this OID in an **AlgorithmIdentifier** shall have the type **EESS1v1-SVES-params**, defined in section 6.2.4 below.

NTRUEncrypt encrypted data should be represented with the **NTRUEncryptedData** type:

```
NTRUEncryptedData ::= NTRUPublicVector
```

The preferred format for **NTRUEncryptedData** is a **ModQVector**.

6.2.4 NTRUEncrypt Parameters

This section defines the parameters associated with the **id-ntru-EESS1v1-SVES** OID in an **AlgorithmIdentifier**. These parameters shall have type **EESS1v1-SVES-Parameters**:

```
EESS1v1-SVES-Parameters ::= CHOICE {
        degree                    Degree, -- this choice is deprecated
        standardNTRUParameters    StandardNTRUParameters,
        explicitNTRUParameters    ExplicitNTRUParameters,
        externalParameters        NULL }

StandardNTRUParameters ::= OIDS.&id({NTRUParameters})

NTRUParameters OIDS ::= {
        { OID id-ees251ep4 } |
        { OID id-ees251ep5 } |
        … -- allows for future expansion
        -- other OIDs defined in previous versions of this standard are deprecated
        }

id-ees251ep4 OBJECT IDENTIFIER ::= {id-eess1-params 12}
id-ees251ep5 OBJECT IDENTIFIER ::= {id-eess1-params 13}
```

- **degree** gives the degree of the polynomials. If this field is specified, it can only take the values 251, 347 or 503. If it is 251, the parameters are eess251ep1. If it is 347, the parameters are eess347ep1. If it is 503, the parameters are eess503ep1. Specifying the degree is the preferred way of transmitting parameter information for this scheme.
- **standardNTRUParameters** identifies the parameters by use of an OID. In this document, six OIDs are defined: **eess139ep1**, **eess139ep2, eess251ep1, eess251ep2**, **eess347ep1** and **eess503ep1**. The parameter sets **eess251ep1**,

eess347ep1 and **eess503ep1** can be specified in this version of the document by choosing to specify the **degree** field instead.

- **explicitNTRUParameters** allows an implementer to specify parameter sets other than those specified in this document. It is not supported in this version of this document.
- **externalParameters** should be used if the parameters are being inherited from some other source (for example, in X.509 certificates, if the parameters are being inherited from the CA's parameters).

6.3 ASN.1 for NTRUSign SVSSA

This section defines the ASN.1 object identifiers for NTRUSign keys and NTRUSign signed data, and defines the types **NTRUSignPublicKey**, **NTRUSignPrivateKey**, **NTRUSignedData**, and **EESS1v1-NTRUSign-Parameters**.

The object identifier **id-ntru-EESS1v1-NTRUSign** identifies NTRUSign public and private keys and NTRUSign signed data. When this object identifier is used in an **AlgorithmIdentifier**, the parameters shall be of type **EESS1v1-NTRUSign-Parameters**.

Note that EESS#1 breaks with common practice in requiring that a key be encoded with the scheme parameters (such as the verification norm bound for NTRUSign) as well as with the algorithm domain parameters (such as N, q and p). Ensuring that a key can only be used in one scheme provides a defense against version rollback attacks and is good security practice.

6.3.1 NTRUSign Public Keys

NTRUSign public keys are identified by the following object identifier:

```
id-ntru-EESS1v1-NTRUSign  OBJECT IDENTIFIER ::=  {id-eess1-algs 3}
```

The parameters associated with this OID in an **AlgorithmIdentifier** shall have the type **EESS1v1-NTRUSign-Parameters**, defined in section 6.3.4 below.

The NTRUSign public key MUST be encoded using the ASN.1 type **NTRUSignPublicKey**.

```
NTRUSignPublicKey ::= SEQUENCE {
    publicKeyVector        NTRUPublicVector,  -- h
    ntruSignKeyExtensions  NTRUSignKeyExtensions OPTIONAL
}

NTRUSignKeyExtensions ::=  SEQUENCE SIZE(1..MAX) OF NTRUSignKeyExtension

NTRUSignKeyExtension ::= CHOICE {
    keyID          [0] IMPLICIT INTEGER,
...}
```

The fields of the type NTRUSignPublicKey have the following meanings:

- **publicKeyVector** is the polynomial h. If the **NTRUPublicVector** is a **ModQVector**, each coefficient will be represented by one byte starting with the lowest degree and going to the highest. If the **NTRUPublicVector** is a **PackedModQVector**, this is the **OCTET STRING** representing h obtained using RE2BSP and then BS2OSP as defined in section 2.1.2. All coefficients up to X^{N-1} SHALL be explicitly included in **publicKeyVector**. Representing the NTRUSign public key as a **ModQVector** is the preferred method.
- **ntruSignKeyExternsions** is provided for future extensibility. Only one extension is currently defined.

The fields of the type NTRUSignKeyExtension have the following meanings:

- **keyID** can be used to associate a unique key identifier with the key.

6.3.2 NTRUSign Private Keys Syntax

NTRUSign private keys are identified by the following object identifier:

id-ntru-EESS1v1-NTRUSign OBJECT IDENTIFIER ::= {id-eess1-algs 3}

They are distinguished from NTRUSign public keys by form and by context. The parameters associated with this OID in an **AlgorithmIdentifier** shall have the type **EESS1v1-NTRUSign-Parameters**, defined in section 6.3.4 below.

The NTRUSign private key should be encoded with the following syntax.

```
NTRUEncryptPrivateKey ::= SEQUENCE {
        version                     INTEGER,
        publicKeyVector             NTRUPublicVector OPTIONAL,
        ntruSignPrivateKeyVectors   NTRUSignPrivateKeyVectors,
        …}

NTRUSignPrivateKeyVectors ::= SEQUENCE {
        mainKey         NTRUSignMainKey,
        perturbations   SEQUENCE OF {
                                NTRUSignPerturbationKey
                        }
        }

NTRUSignMainKey ::= SEQUENCE {
        f               NTRUSignKeyVector
        fPrime          NTRUSignKeyVector
}

NTRUSignPerturbationKey ::= SEQUENCE {
        f               NTRUSignKeyVector
        fPrime          NTRUSignKeyVector
        h1minusH        NTRUPublicVector
}
```

```
NTRUSignKeyVector ::= CHOICE {
        listedBinaryVector      [0] IMPLICIT ListedBinaryVector,
        packedBinaryVector      [1] IMPLICIT PackedBinaryVector,
        productFormKey          [2] IMPLICIT ProductFormKey,
        modQVector              [3] IMPLICIT ModQVector,
        packedModQVector        [4] IMPLICIT PackedModQVector
}

ProductFormKey ::= SEQUENCE {
        a1      ListedBinaryVector,
        a2      ListedBinaryVector,
        a3      ListedBinaryVector,
}
-- product form keys are of the form a1*a2+a3
```

The fields of the type **NTRUPrivateKey** have the following meanings:

- **version** is the version number, for compatibility with future revisions of this document. It shall be 1 for this version of the document. Previous versions of this document used version number 0; this version is no longer supported.
- **publicKeyVector** is the public key associated with the private key. It can be provided either explicitly in this field, or implicitly by providing the **GVectors** in the **ntruPrivateKeyVectors** field.
- **ntruSignPrivateKeyVectors** contains the private key vectors as specified below..

The fields of the type **NTRUSignPrivateKeyVectors** have the following meanings:

- **mainKey** contains the polynomials f_0, f'_0, as output by the key generation primitive.
- **perturbations** contains the perturbation keys (f_1, f'_1) … (f_B, f'_B), in that order. The number of perturbations is not explicitly stored, but can be deduced from the number of **NTRUSignPerturbationKey**s contained in **perturbations**.

The fields of the type **NTRUSignMainKey** have the following meanings:

- **f** contains the polynomial f_0, as output by the key generation primitive. For the parameter sets in this standard, f_0 will be binary and the preferred encoding of this polynomial is as a **PackedBinaryVector**.
- **f** contains the polynomial f'_0, as output by the key generation primitive. For the parameter sets **ees251sp2** – **ees251sp5** in this standard, f'_0 will be a mod q polynomial and the preferred encoding of this polynomial is as a **ModQVector**. For the parameter sets **ees251sp6** – **ees251sp9** in this standard, f'_0 will be a binary polynomial and the preferred encoding of this polynomial is as a **PackedBinaryVector**.

The fields of the type **NTRUSignPerturbationKey** have the following meanings for perturbation key i:

- **f** contains the polynomial f_i, as output by the key generation primitive. For the parameter sets in this standard, f_i will be binary and the preferred encoding of this polynomial is as a **PackedBinaryVector**.
- **f** contains the polynomial f'_i, as output by the key generation primitive. For the parameter sets **ees251sp6** – **ees251sp9** in this standard, f'_0 will be a binary polynomial and the preferred encoding of this polynomial is as a **PackedBinaryVector**. (For the parameter sets **ees251sp2** – **ees251sp5** in this standard, there are no perturbation keys).
- **h1minush** contains the polynomial h_i – h_{i-1}, as output by the key generation primitive.

The fields of the type **NTRUSignKeyVectors** have the following meanings:

- **listedBinaryVector** contains the relevant polynomial encoded as a **ListedBinaryVector**.
- **packedBinaryVecto** contains the relevant polynomial encoded as a **PackedBinaryVector**.
- **productFormKey** contains the relevant polynomial encoded as a **ProductFormKey**.
- **modQVector** contains the relevant polynomial encoded as a **ModQVector**.
- **packedModQVector** contains the relevant polynomial encoded as a **PackedModQVector**.

6.3.3 NTRUSign Signed Data

NTRUSign signed data are identified by the following object identifier:

id-ntru-EESS1v1-NTRUSign OBJECT IDENTIFIER ::= {id-eess1-algs 3}

The parameters field associated with this OID in an **AlgorithmIdentifier** shall have the type **EESS1v1-NTRUSign-Parameters**, defined in section 6.3.4 below.

NTRUSign signed data should be represented with the **NTRUSignSignededData** type:

NTRUSignSignedData ::= NTRUPublicVector

The preferred format for **NTRUSignSignedData** is a **ModQVector**.

6.3.4 NTRUSign Parameters

This section defined the parameters associated with the **id-ntru-EESS1v1-NTRUSign** OID in an **AlgorithmIdentifier**. These parameters shall have type **EESS1v1-NTRUSign-Parameters**:

```
EESS1v1-NTRUSign-Parameters ::= CHOICE {
    degree                              Degree, -- deprecated
    standardNTRsUSignParameters                 StandardNTRUSignParameters,
    explicitNTRUSignParameters          ExplicitNTRUSignParameters,
    externalParameters                  NULL
}
```

When the parameters are implied by context, the parameters field SHALL contain **externalParameters**, which is the ASN.1 value NULL.

When the parameters are specified by reference of a standard, the parameters shall consist of an OID chosen from the list **NTRUSignParameters**. The current list of **NTRUSignParameters** OIDs is:

```
StandardNTRUSignParameters ::= OIDS.&id({NTRUSignParameters})

NTRUSignParameters  OIDS ::= {
      { OID id-ees251sp2 },
      { OID id-ees251sp3 },
      { OID id-ees251sp4 },
      { OID id-ees251sp5 },
      { OID id-ees251sp6 },
      { OID id-ees251sp7 },
      { OID id-ees251sp8 },
      { OID id-ees251sp9 },
...}
```

The above object identifiers are specified by:

```
id-ees251sp2  OBJECT IDENTIFIER ::= {id-eess1-params 7}
id-ees251sp3  OBJECT IDENTIFIER ::= {id-eess1-params 14}
id-ees251sp4  OBJECT IDENTIFIER ::= {id-eess1-params 15}
id-ees251sp5  OBJECT IDENTIFIER ::= {id-eess1-params 16}
id-ees251sp6  OBJECT IDENTIFIER ::= {id-eess1-params 17}
id-ees251sp7  OBJECT IDENTIFIER ::= {id-eess1-params 18}
id-ees251sp8  OBJECT IDENTIFIER ::= {id-eess1-params 19}
id-ees251sp9  OBJECT IDENTIFIER ::= {id-eess1-params 20}
```

When the parameters are explicitly included, they SHALL be encoded in the ASN.1 structure **ExplicitNTRUSignParameters**. This structure is not supported in this version of this standard.

6.4 X.509 Certificates

```
-- The following section is written in ASN.1 notation.  This section specifies
-- the basic ASN.1 structure for EESS X.509 certificates.  Embedded within the
-- normal X.509 data are comments indicating the preference for EESS certificates.
-- The X.509 structure below is written to be compliant with the current draft of the
-- ITU-T recommendation [X.509] and the IETF PKIX ID son-of-rfc 2459
-- [ID son-of-rfc2459].
-- These formats are recommended for interoperability, but are not mandated by this
-- standard.

Certificate  ::=  SEQUENCE  {
      tbsCertificate       TBSCertificate,
      signatureAlgorithm   AlgorithmIdentifier,
      signatureValue       BIT STRING
}
```

```
  TBSCertificate  ::=  SEQUENCE  {
     version         [0]  EXPLICIT Version DEFAULT v1,
     serialNumber         CertificateSerialNumber,
     signature            AlgorithmIdentifier,
     issuer               Name,
     validity             Validity,
     subject              Name,
     subjectPublicKeyInfo SubjectPublicKeyInfo,
     issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                          -- If present, version shall be v2 or v3
                        -- The issuerUniqueID is deprecated and
                        -- should not be used for EESS certificates.
     subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                          -- If present, version shall be v2 or v3
                        -- The subjectUniqueID is deprecated and
                        -- should not be used for EESS certificates.
    extensions      [3]  EXPLICIT Extensions OPTIONAL
                          -- If present, version shall be v3
                        -- Extensions are expected in EESS certificates.
     }

  Version  ::=  INTEGER  {  v1(0), v2(1), v3(2)  }
                        -- EESS certificates shall be version 3.

  CertificateSerialNumber  ::=  INTEGER
                        -- CertificateSerialNumber is not restricted and is
                        -- implementation specific.

  AlgorithmIdentifier ::= SEQUENCE {
     algorithm      OBJECT IDENTIFIER,
     parameters       ANY DEFINED BY algorithm OPTIONAL }
                        -- The algorithm identifier for NTRUSign signatures is specified
                        -- in sections 6.3.3.

Name ::= CHOICE {
   RDNSequence }
                        -- The CHOICE shall be RDNSequence

  RDNSequence ::= SEQUENCE OF RelativeDistinguishedName
                        -- There are no restrictions on the number of
                        -- RelativeDistinguishedNames in a name.

  RelativeDistinguishedName ::= SET OF AttributeTypeAndValue
                        -- There is almost always one AttributeTypeAndValue
                        -- per RelativeDistinguishedName

  AttributeTypeAndValue ::= SEQUENCE {
   type     AttributeType,
   value    AttributeValue }

  AttributeType ::= OBJECT IDENTIFIER
  AttributeValue ::= ANY DEFINED BY AttributeType
                        -- Where there is a DirectoryString, the preferred type is
                        -- UTF8String.  In [ID son-of-rfc2459]], it is stated that after
                        -- December 31, 2003, all DirectoryStrings must be UTF8String encoded.
```

```
 Validity ::= SEQUENCE {
      notBefore      Time,
      notAfter       Time }
                    -- The validity period is application specific and
                    -- not specified in this document.

  Time ::= CHOICE {
      utcTime        UTCTime,
      generalTime    GeneralizedTime }
                    -- Following the rules for UTCTime and GeneralizedTime,
                    -- the choice will be UTCTime for all years through 2049.
                    -- The time value should be encoded as (YYMMDDHHMMSSZ),
                    -- where Z stands for GMT.

UniqueIdentifier  ::=  BIT STRING
                    -- This shall not be used.

SubjectPublicKeyInfo  ::=  SEQUENCE  {
      algorithm            AlgorithmIdentifier,
      subjectPublicKey     BIT STRING  }
                    -- See sections 6.2.1 and 6.3.1 for NTRUEncrypt and NTRUSign definitions.

Extensions  ::=  SEQUENCE SIZE (1..MAX) OF Extension

Extension  ::=  SEQUENCE  {
      extnID      OBJECT IDENTIFIER,
      critical    BOOLEAN DEFAULT FALSE,
      extnValue   OCTET STRING  }
```

Appendix A - NTRU ASN.1 Module

```
EESS-1 {iso(1) identified-organization(3) dod(6) internet(1) private(4) enterprises(1)
ntruCryptosystems(8342) eess(1) eess-1(1) modules(0) eess-1(1)}

 -- $ revision: 2.0 $

DEFINITIONS EXPLICIT TAGS ::= BEGIN

-- EXPORTS All; --
-- All types and values defined in this module are exported for use in other ASN.1 modules.

-- IMPORTS None; --

-- Supporting definitions

AlgorithmIdentifier { ALGORITHM: IOSet } ::= SEQUENCE {
        algorithm       ALGORITHM.&id({IOSet}),
        parameters      ALGORITHM.&Type({IOSet}{@algorithm}) OPTIONAL
}

ALGORITHM ::= CLASS {
        &id     OBJECT IDENTIFIER  UNIQUE,
        &Type  OPTIONAL
}
        WITH SYNTAX { OID &id [PARMS &Type] }

OIDS ::= ALGORITHM

-- Informational object identifiers

pkcs-1 OBJECT IDENTIFIER ::= {
        iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) 1 }

id-mgf1 OBJECT IDENTIFIER ::= { pkcs-1 8}

id-sha1 OBJECT IDENTIFIER ::= {
        iso(1) identified-organization(3) oiw(14) secsig(3) algorithms(2) 26 }

id-sha256 OBJECT IDENTIFIER ::= {
        joint-iso-itu-t(2) country(16) us(840) organization(1) gov(101)
        csor(3)  nistalgorithm(4) hashalgs(2) 1 };

id-sha384 OBJECT IDENTIFIER ::= {
        joint-iso-itu-t(2) country(16) us(840) organization(1) gov(101)
        csor(3)  nistalgorithm(4) hashalgs(2) 2 };

id-sha512 OBJECT IDENTIFIER ::= {
        joint-iso-itu-t(2) country(16) us(840) organization(1) gov(101)
        csor(3)  nistalgorithm(4) hashalgs(2) 3 };


-- Basic object identifiers
```

```
ntru OBJECT IDENTIFIER ::= {
        iso(1) identified-organization(3) dod(6) internet(1) private(4) enterprises(1)
        ntruCryptosystems (8342) }

        id-eess1 OBJECT IDENTIFIER ::= { ntru eess (1) eess-1 (1) }

id-eess1-algs OBJECT IDENTIFIER ::= {id-eess1  1}
id-eess1-params OBJECT IDENTIFIER ::= {id-eess1  2}
id-eess1-encodingMethods OBJECT IDENTIFIER ::= {id-eess1 3}

-- algorithms

id-ntru-EESS1v1-SVES OBJECT IDENTIFIER  ::= {id-eess1-algs  1}
id-ntru-EESS1v1-NTRUSign OBJECT IDENTIFIER  ::= {id-eess1-algs 3}
id-mdc-2des-ntru OBJECT IDENTIFIER ::= {id-eess1-algs 4}

-- parameter set identifiers

id-ees251ep4 OBJECT IDENTIFIER ::= {id-eess1-params 12}
id-ees251ep5 OBJECT IDENTIFIER ::= {id-eess1-params 13}

id-ees251sp2  OBJECT IDENTIFIER ::= {id-eess1-params 7}
id-ees251sp3  OBJECT IDENTIFIER ::= {id-eess1-params 14}
id-ees251sp4  OBJECT IDENTIFIER ::= {id-eess1-params 15}
id-ees251sp5  OBJECT IDENTIFIER ::= {id-eess1-params 16}
id-ees251sp6  OBJECT IDENTIFIER ::= {id-eess1-params 17}
id-ees251sp7  OBJECT IDENTIFIER ::= {id-eess1-params 18}
id-ees251sp8  OBJECT IDENTIFIER ::= {id-eess1-params 19}
id-ees251sp9  OBJECT IDENTIFIER ::= {id-eess1-params 20}

-- General types

ModQVector ::= OCTET STRING

PackedModQVector ::= OCTET STRING

ListedBinaryVector ::= OCTET STRING

PackedBinaryVector ::= OCTET STRING

NTRUGeneralPolynomial ::= SEQUENCE {
        numberOfEntries              INTEGER,
        modulus                      INTEGER,
        coefficients                 GeneralVector
}

GeneralVector ::= OCTET STRING

-- NTRUEncrypt Encryption

NTRUPublicVector ::= CHOICE {
        modQVector           [0] IMPLICIT ModQVector,
        packedModQVector     [1] IMPLICIT PackedModQVector,
        …
}
```

```
NTRUBinaryVector ::= CHOICE {
        listedBinaryVector      [0] IMPLICIT ListedBinaryVector,
        packedBinaryVector      [1] IMPLICIT PackedBinaryVector,
        modQVector              [2] IMPLICIT ModQVector,
        …
}

NTRUKeyExtension ::= CHOICE {
        keyID           [0] IMPLICIT INTEGER,
        …}

NTRUPublicKey ::= SEQUENCE {
        publicKeyVector  NTRUPublicVector,
        ntruKeyExtensions       SEQUENCE SIZE (1..MAX) OF
                        NTRUKeyExtension OPTIONAL }

PrivateKeyType ::= INTEGER
FVectors ::= SEQUENCE OF NTRUBinaryVector
GVectors ::= SEQUENCE OF NTRUBinaryVector

NTRUPrivateKeyVectors ::= SEQUENCE {
        fVectors        FVectors,
        gVectors        [0] IMPLICIT GVectors OPTIONAL }

NTRUPrivateKey ::= SEQUENCE {
        version                 INTEGER,
        publicKeyVector         NTRUPublicVector OPTIONAL,
        privateKeyType          PrivateKeyType,
        ntruPrivateKeyVectorsNTRUPrivateKeyVectors,
        …}

NTRUEncryptedData ::= NTRUPublicVector

Degree ::= INTEGER  (251 | 347 | 503, …)

NTRUParameters OIDS ::= {
        { OID id-ees251ep4 } |
        { OID id-ees251ep5 } |
        … -- allows for future expansion
        -- other OIDs defined in previous versions of this standard are deprecated
        }

Version ::= INTEGER { v0(0) } (v0, …)

-- NTRUSign Signing

-- Encoding for NTRUSign Signatures

 NTRUSignSignedData ::= NTRUPublicVector

-- Encoding for NTRUSign Public Keys

 NTRUSignPublicKey ::= SEQUENCE {
      publicKeyVector        NTRUPublicVector,  -- h
      ntruSignKeyExtensions   NTRUSignKeyExtensions OPTIONAL
```

```
       }

 NTRUSignKeyExtensions ::=
       SEQUENCE SIZE(1..MAX) OF NTRUSignKeyExtension

 NTRUSignKeyExtension ::= CHOICE {
       keyID          [0] IMPLICIT INTEGER,
       ...}

NTRUEncryptPrivateKey ::= SEQUENCE {
          version                         INTEGER,
          publicKeyVector                 NTRUPublicVector OPTIONAL,
          ntruSignPrivateKeyVectors       NTRUSignPrivateKeyVectors,
          …}

NTRUSignPrivateKeyVectors ::= SEQUENCE {
          mainKey          NTRUSignMainKey,
          perturbations    SEQUENCE OF {
                                    NTRUSignPerturbationKey
                           }
         }

NTRUSignMainKey ::= SEQUENCE {
          f                NTRUSignKeyVector
          fPrime           NTRUSignKeyVector
}

NTRUSignPerturbationKey ::= SEQUENCE {
          f                NTRUSignKeyVector
          fPrime           NTRUSignKeyVector
          h1minusH         NTRUPublicVector
}

NTRUSignKeyVector ::= CHOICE {
          listedBinaryVector      [0] IMPLICIT ListedBinaryVector,
          packedBinaryVector      [1] IMPLICIT PackedBinaryVector,
          productFormKey          [2] IMPLICIT ProductFormKey,
          modQVector              [3] IMPLICIT ModQVector,
          packedModQVector        [4] IMPLICIT PackedModQVector
}

ProductFormKey ::= SEQUENCE {
          a1        ListedBinaryVector,
          a2        ListedBinaryVector,
          a3        ListedBinaryVector,
}
-- product form keys are of the form a1*a2+a3

END  -- EESS-1 --
```

Appendix B - Test Vectors

[To be added in future versions]

Appendix C - Revision History

Draft 1.0 available March 27, 2001
Draft 2.0 available May 18, 2001
Draft 3.0 available July 9, 2001
Draft 3.2 available August 30, 2001
Draft 4.0 available March 9, 2002
Draft 5.0 available September 6, 2002
Version 1.0 available November 12, 2002
Version 2.0 available April 14, 2003

Appendix D - References

[ANS98a] ANSI X9.31-1998, Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry (rDSA).

[ANS98b] ANSI X9.42, Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Diffie-Hellman and MQV Algorithms, draft, 1998.

[ANS98c] ANSI X9.62-1998, Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA).

[FIP95] FIPS PUB 180-1, Secure Hash Standard, Federal Information Processing Standards Publication 180-1, U.S. Department of Commerce/National Institute of Standards and Technology, National Technical Information Service, Springfield, Virginia, April 17, 1995 (supersedes FIPS PUB 180). Available at http://www.itl.nist.gov/div897/pubs/fip180-1.htm.

[FIP99] FIPS PUB 46-3, Data Encryption Standard, Federal Information Processing Standards Publication 186-2, U.S. Department of Commerce/National Institute of Standards and Technology, National Technical Information Service, Springfield, Virginia, October 1999. Available at http://csrc.nist.gov/fips/.

[FIP00] FIPS PUB 186-2, Digital Signature Standard, Federal Information Processing Standards Publication 186-2, U.S. Department of Commerce/National Institute of Standards and Technology, National Technical Information Service, Springfield, Virginia, February 2000. Available at http://csrc.nist.gov/fips/.

[HPS98] J. Hoffstein, J. Pipher, J. Silverman, NTRU: A Ring Based Public Key Cryptosystem, *Algorithmic Number Theory (ANTS III)*, Portland, OR, June 1998, J.*P.* Buhler (ed.), Lecture Notes in Computer Science 1423, Springer-Verlag, Berlin, 1998, 267-288.

[HPS01] J. Hoffstein, J. Pipher, J. Silverman, NSS: The NTRU Signature Scheme, *Proc. EUROCRYPT 2001*, Lecture Notes in Computer Science, Springer-Verlag,, 2001, to appear.

[HPS01-2] J. Hoffstein, J. Pipher, J. Silverman, Enhanced Encoding and Verification Methods for the NTRU Signature Scheme, NTRU Technical Report 017, 2001, <www.ntru.com>.

[HS99] J. Hoffstein, J. Silverman, Reaction attacks against the NTRU public key cryptosystem, NTRU Technical Report 015, 1999, <www.ntru.com>.

[HS00] J. Hoffstein, J. Silverman, Optimizations for NTRU, *Public-Key Cryptography and Computational Number Theory* (Warsaw, September 2000), DeGruyter, to appear.

[HS00-2] J. Hoffstein, J. Silverman, Protecting NTRU Against Chosen Ciphertext and Reaction Attacks, Technical Report 16, 2000, <www.ntru.com>.

[HS01] J. Hoffstein, J. Silverman, Random Small Hamming Weight Products with Applications to Cryptography, Com2MaC Workshop on Cryptography (Pohang, Korea, June 2000), *Discrete Mathematics*, to appear.

[IEEE 1363] IEEE Std 1363-2000: IEEE Standard Specifications for Public-Key Cryptography, IEEE Computer Society, New York, NY, August 2000, Institute of Electrical and Electronics Engineers

[IEEE P1363.1] IEEE Draft Standard P1363.1 D2: IEEE Standard Specifications for Public-Key Cryptographic Techniques Based on Hard Problems over Lattices, Draft 2, May 2001, Available at http://grouper.ieee.org/groups/1363.

[ITU-T X.509] ITU-T Recommendation X.509 (pre-published 03/00) (also ISO/IEC 9594-8:1998): Information Technology – Open Systems Interconnection – The Directory: Authentication Framework

[ID son-of-rfc2459] draft-ietf-pkix-new-part1-03.txt – Internet X.509 Public Key Infrastructure Certificate and CRL Profile (to obsolete RFC 2459), Housley et al, November 2000

[NIST-SHA-2] National Institute of Standards and Technology. Descriptions of SHA-256, SHA-384, and SHA-512. October 12, 2000. Available at http://csrc.nist.gov/cryptval/shs.html.

2.2.6 2.2.7 2.3.1 2.3.2 2.3.3 2.3.4 2.3.5 2.3.6 3.2.1 3.2.1.1 3.2.2 3.2.2.1 3.2.3 3.2.4 **Error! Reference source not found.** 3.3.1 3.3.1.1 3.5.1.1 3.5.2.1 3.6.1 3.6.1.1 3.6.1.2 3.7.1 3.7.1.1 3.7.2 3.7.2.1 3.7.2.2 3.7.3 3.7.3.1 3.7.3.2 3.7.4 4.3