Efficient Embedded Security Standards (EESS)

EESS #1: Implementation Aspects of NTRUEncrypt

Consortium for Efficient Embedded Security

March 31st, 2015

Version 3.0

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

Table of Contents

1 Overview

1.1 Scope

This document specifies common techniques and implementation choices using the NTRUEncrypt 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
- 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, including security.

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. This document will be regularly updated via the NTRU Open Source repository on github, https://github.com/NTRUOpenSourceProject/ntru-crypto, and other means as appropriate. In addition, the Consortium welcomes input from the community at large. Comments may be submitted to the editor, William Whyte, at wwhyte@securityinnovation.com.

1.5 Intellectual Property

Some of the techniques in this standard are the subject of intellectual property owned by Security Innovation. This intellectual property may be made use of under the license terms available from https://github.com/NTRUOpenSourceProject/ntru-crypto.

2 Normative references

The following referenced documents are indispensable for the application of this document (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

1. FIPS 180, *Secure Hash Standard*, Federal Information Processing Standards Publication 180, U.S. Department of Commerce/National Institute of Standards and Technology, National Technical Information Service, Springfield, Virginia.[1]

3 Definitions, acronyms and abbreviations

3.1 Definitions

For the purposes of this standard, the following terms and definitions apply.

3.1.1 **Algorithm:** A clearly specified mathematical process for computation; a set of rules which, if followed, give a prescribed result.

3.1.2 **Asymmetric Cryptographic Algorithm:** A cryptographic algorithm that uses two related keys, a public key and a private key; the two keys have the property that, given the public key, it is computationally infeasible to derive the private key.

3.1.3 **Authentication (of a message):** The act of determining that a message has not been changed since leaving its point of origin. The identity of the originator is implicitly verified.

3.1.4 **Authentication of Ownership:** The assurance that a given, identified party intends to be associated with a given public key. May also include assurance that the party possesses the corresponding private key (see IEEE Std 1363-2000, Annex D.3.2, for more information).

3.1.5 **Big Modulus:** The big modulus q is used to define the larger polynomial ring. The modulus q can generally be taken to be any value that is relatively prime in the ring to the small modulus p.

3.1.6 **Birthday Paradox:** For a category size of 365 (the days in a year), after only 23 people are gathered, the probability is greater than 0.5 that at least two people have a common birthday (month and day). The reason is that among 23 people, there are $23 \times (23\text{-}1)/2 = 253$ pairs of people, each with a 1/365 chance of having matching birthdays. The chance of no matching birthday is therefore $(364/365)^{253} \sim 0.4995$. In general, any case where the criterion for success is to find a collision (two matching values) rather than a hit (one value which matches a pre-selected one) displays this pairing property, so that the size of the space to be searched for success is about the square root of the size of the space of all possible values.

3.1.7 **Bit Length:** See: length.

3.1.8 **Bit String:** An ordered sequence of 0's and 1's. The left-most bit is the most-significant bit of the string. The right-most bit is the least-significant bit of the string. A bit and a bit string of length 1 are equivalent for all purposes of this standard.

[1] FIPS 180 current version as of 2015 is FIPS 180-4, March, 2012, available at http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf.

3.1.9 Blinding Polynomial: In this standard, the ciphertext *e* is generated according to the equation $e = r \times h + m'$, where *h* is the public key, *m*' is the message representative, and r is a pseudorandomly generated "blinding polynomial"

3.1.10 Blinding Polynomial Generation Methods: In the encryption schemes in this document, a blinding polynomial generation method (LBP-BPGM) is used to generate a blinding polynomial r from the padded message pm in order to provide plaintext awareness.

3.1.11 Blinding Polynomial Space: The space that a LBP-BPGM selects from. Usually defined implicitly by the definition of the LBP-BPGM.

3.1.12 Certificate: The public key and identity of an entity together with some other information rendered unforgeable by signing the certificate with the private key of the certifying authority, which issued that certificate.

3.1.13 Ciphertext: The result of applying encryption to a message. Contrast: plaintext. See also: encryption.

3.1.14 Composite: An integer which has at least two prime factors.

3.1.15 Confidentiality: The property that information is not made available or disclosed to unauthorized individuals, entities, or processes.

3.1.16 Conformance Region: a set of inputs to a primitive or a scheme operation for which an implementation operates in accordance with the specification of the primitive or scheme operation

3.1.17 Cryptographic Family: A set of cryptographic techniques in similar mathematical settings. For example, this standard presents a single family of techniques based on the underlying hard problems of finding a short vector and a close vector in a lattice.

3.1.18 Cryptographic Hash Function: See hash function.

3.1.19 Cryptographic Key (Key): A parameter that determines the operation of a cryptographic function such as: the transformation from plain text to cipher text and vice versa; synchronized generation of keying material; digital signature computation or validation.

3.1.20 Cryptography: The discipline which embodies principles, means and methods for the transformation of data in order to hide its information content, prevent its undetected modification, prevent its unauthorized use or a combination thereof.

3.1.21 Data Integrity: A property whereby data has not been altered or destroyed.

3.1.22 Decrypt: To produce plaintext (readable) from ciphertext (unreadable). Contrast: encrypt. See also: ciphertext; encryption; plaintext.

3.1.23 Dimension: The dimension *N* identifies the dimension of the convolution polynomial ring used. The dimension of the associated lattice problem is $2N$. Elements of the ring are represented as polynomials of degree $N - 1$.

3.1.24 Domain Parameters: a set of mathematical objects, such as fields or groups, and other information, defining the context in which public/private key pairs exist. More than one key pair may share the same domain parameters. Not all cryptographic families have domain parameters. See also: public/private key pair; valid domain parameters.

3.1.25 Domain Parameter Validation: the process of ensuring or verifying that a set of domain parameters is valid. See also: domain parameters; key validation; valid domain parameters.

3.1.26 Encrypt: to produce ciphertext (unreadable) from plaintext (readable). Contrast: decrypt. See also: ciphertext; encryption; plaintext.

3.1.27 Encryption Primitives: An operation that converts a plaintext to a ciphertext, providing security according to the difficulty of solving an underlying hard problem, against a ciphertext-only attack by a passive attacker who only has a single non-chosen ciphertext. A building block for encryption schemes.

3.1.28 Encryption Scheme: A means for providing encryption, based on an encryption primitive, that is secure against both active and passive attackers. A secure encryption scheme typically provides semantic security (an attacker who knows that one of two messages has been encrypted will find it computationally infeasible to determine which) against an attacker who can make polynomially many queries to a decryption oracle.

3.1.29 Entity: A participant in any of the schemes in this standard. The words "entity" and "party" are used interchangeably. This definition may admit many interpretations: it may or may not be limited to the necessary computational elements; it may or may not include or act on behalf of a legal entity. The particular interpretation chosen does not affect operation of the key agreement schemes.

3.1.30 Exclusive OR: A mathematical bit-wise operation, symbol $\oplus$, defined as:

$0 \oplus 0 = 0$,
$0 \oplus 1 = 1$,
$1 \oplus 0 = 1$, and
$1 \oplus 1 = 0$.

Equivalent to binary addition without carry. May also be applied to bit strings: the XOR of two bit strings of equal length is the concatenation of the XORs of the corresponding elements of the bit strings.

3.1.31 Family: See: cryptographic family.

3.1.32 Field: A setting in which the usual mathematical operations (addition, subtraction, multiplication, and division by nonzero quantities) are possible and obey the usual rules (such as the commutative, associative, and distributive laws).

3.1.33 Finite Field: a field in which there are only a finite number of quantities.

3.1.34 First Bit: the leading bit of a bit string or an octet. For example, the first bit of 0110111 is 0. Contrast: last bit. Syn: most significant bit; leftmost bit. See also: bit string; octet.

3.1.35 First Octet: the leading octet of an octet string. For example, the first octet of 1c 76 3b e4 is 1c. Contrast: last octet. Syn: most significant octet; leftmost octet. See also: octet; octet string.

3.1.36 Hash Function: A function which maps a bit string of arbitrary length to a fixed-length bit string and satisfies the following properties:

1. It is computationally infeasible to find any input which maps to any pre-specified output;
2. It is computationally infeasible to find any two distinct inputs which map to the same output.

3.1.37 Hash Value: The result of applying a hash function to a message.

3.1.38 Index Generation Function: An IGF is a function that is seeded once, can be called multiple times, and produces statistically independent integers modulo some number *m* on each call.

3.1.39 Key: See cryptographic key.

3.1.40 Key Confirmation: The assurance of the legitimate participants in a key establishment protocol that the intended recipients of the shared key actually possess the shared key.

3.1.41 Key Derivation: The process of deriving one or more session keys from a shared secret and (possibly) other, public information. Such a function can be constructed from a one-way hash function such as SHA-1.

3.1.42 Key Encrypting Key (KK): A key used exclusively to encrypt and decrypt keys.

3.1.43 Key Establishment: A protocol that reveals a secret key to its legitimate participants for cryptographic use.

3.1.44 Key Generation Primitive: A method used to generate a key pair.

3.1.45 Key Management: The generation, storage, secure distribution and application of keying material in accordance with a security policy.

3.1.46 Key Pair: When used in public key cryptography, a private key and its corresponding public key. The public key is commonly available to a wide audience and can be used to encrypt messages or verify digital signatures; the private key is held by one entity and not revealed to anyone--it is used to decrypt messages encrypted with the public key and/or produce signatures that can verified with the public key. A public/private key pair can also be used in key agreement. In some cases, a public/private key pair can only exist in the context of domain parameters. See also: digital signature; domain parameters; encryption; key agreement; public-key cryptography; valid key; valid key pair.

3.1.47 Key Transport: A key establishment protocol under which the secret key is determined by the initiating party.

3.1.48 Key Validation: the process of ensuring or verifying that a key conforms to the arithmetic requirements for such a key in order to thwart certain types of attacks. See also: domain parameter validation; public/private key pair; valid key; valid key pair.

3.1.49 Keying Material: The data (e.g., keys, certificates and initialization vectors) necessary to establish and maintain cryptographic keying relationships.

3.1.50 Known-Key Security: Known-key security for Party U implies that the key agreed upon will not be compromised by the compromise of the other session keys. If each ephemeral key is used only to compute a single session key, then known-key security may be achieved.

3.1.51 Last Bit: The trailing bit of a bit string or an octet. For example, the last bit of 0110111 is 1. Contrast: first bit. Syn: least significant bit; rightmost bit. See also: first bit; octet.

3.1.52 Last Octet: The trailing octet of an octet string. For example, the last octet of 1c 76 3b e4 is e4. Contrast: first octet. Syn: least significant octet; rightmost octet. See also: octet; octet string.

3.1.53 Lattice Based Polynomial Public Key Encryption: An encryption mechanism where operations are based on polynomial multiplication and the security is based on the difficulty of performing high-dimension lattice reduction.

3.1.54 Least Significant: See: last bit; last octet.

3.1.55 Leftmost Bit: See: first bit.

3.1.56 Leftmost Octet: See: first octet.

3.1.57 Length: (1) Length of a bit string is the number of bits in the string. (2) Length of an octet string is the number of octets in the string. (3) Length in bits of a nonnegative integer n is $\lfloor \log2\ (n + 1) \rfloor$ (i.e., the number of bits in the integer's binary representation). (4) Length in octets of a nonnegative integer n is $\lfloor \log256\ (n + 1) \rfloor$ (i.e., the number of digits in the integer's representation base 256). For example, the length in bits of the integer 500 is 9, and its length in octets is 2.

3.1.58 Mask Generation Function: An MGF is a construction built around a hash function that produces an arbitrary-length output string, possibly longer than the output of the underlying hash function.

3.1.59 Message Authentication Code (MAC): A cryptographic value which is the results of passing a financial message through the message authentication algorithm using a specific key.

3.1.60 Message Length Encoding Length: In SVES, the length of the message that is to be encrypted is encoded in the padded message. 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 all parameter sets in this document *lLen* is set to 1.

3.1.61 Message Representative: A mathematical value for use in a cryptographic primitive, computed from a message that is input to an encryption or a digital signature scheme and uniquely linked to that message. See also: encryption scheme; digital signature scheme.

3.1.62 Modular Lattice: A lattice in which (among other things) all values are integers reduced mod *q*.

3.1.63 Most Significant: See: first bit; first octet.

3.1.64 Norm: A measure of the "size" of a vector or polynomial.

3.1.65 Octet: A bit string of length 8. An octet has an integer value between 0 and 255 when interpreted as a representation of an integer in base 2. An octet can also be represented by a hexadecimal string of length 2, where the hexadecimal string is the representation of its integer value base 16. For example, the integer value of the octet 10011101 is 157; its hexadecimal representation is 9d. Also commonly known as a byte. See also: bit string.

3.1.66 Octet String: An ordered sequence of octets. See also: octet.

3.1.67 Owner: The entity whose identity is associated with a key pair.

3.1.68 Parameters: See: domain parameters.

3.1.69 Plaintext: A message before encryption has been applied to it; the opposite of ciphertext. Contrast: ciphertext. See also: encryption.

3.1.70 Polynomial Index Generation Constant: A value used when generating a random number in the range [0, *N*-1], to eliminate bias without impacting efficiency.

3.1.71 Prime Number: An integer that is greater than 1 and divisible only by 1 and itself.

3.1.72 Primitives: Cryptographic primitives used in the SVES encryption scheme include key generation primitives, encryption primitives and decryption primitives.

3.1.73 Private Key: The private element of the public/private key pair. See also: public/private key pair; valid key.

3.1.74 Private Key Space: The space from which a key generation primitive selects the private key.

3.1.75 Public Key: The public element of the public/private key pair. See also: public/private key pair; valid key.

3.1.76 Public-key Cryptography: methods that allow parties to communicate securely without having prior shared secrets through the use of public/private key pairs. Contrast: symmetric cryptography. See also: public/private key pair.

3.1.77 Public Key Space: The space from which a key generation primitive selects the public key.

3.1.78 Public Key Validation: See key validation.

3.1.79 Public/Private Key Pair: See key pair.

3.1.80 Salt: Random bits that are used to pad the message during encryption, to provide for semantic security.

3.1.81 Salt Size: The size of the salt. Can be expressed in bits or octets.

3.1.82 Rightmost Bit: See: last bit.

3.1.83 Rightmost Octet: See: last octet.

3.1.84 Ring: a setting in which addition, subtraction, and multiplication are possible, and division by a given nonzero quantity may or may not be possible. A field is a special case of a ring. See also: field.

3.1.85 Ring Element: in general, an element in a ring. In the context of this standard, a *binary N-ring element* refers to an element in the ring $(\mathbf{Z}/2\mathbf{Z})[X]/(X^N - 1)$, which is to say a binary polynomial of degree *N*-1 or an array of *N* binary elements. A (*q*, *N*)-*ring element* refers to an element in the ring $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$, which is to say a polynomial of degree *N*-1 with coefficients reduced mod *q* or an array of *N* elements each taken mod *q*.

3.1.86 Scheme Options: Scheme options consist of parameters and algorithms that do not affect the key space (i.e. that are not domain parameters), but that shall be agreed upon in order to implement the encryption scheme.

3.1.87 Secret Key: a key used in symmetric cryptography; needs to be known to all legitimate participating parties involved, but cannot be known to an adversary. Contrast: public/private key pair. See also: key agreement; shared secret key; symmetric cryptography.

3.1.88 Secret Value: a value that can be used to derive a secret key, but typically cannot by itself be used as a secret key. See also: secret key.

3.1.89 Shared Secret Key: a secret key shared by two parties, usually derived as a result of a key agreement scheme. See also: key agreement; secret key.

3.1.90 Shared Secret Value: a secret value shared by two parties, usually during a key agreement scheme. See also: key agreement; secret value.

3.1.91 Signature: See: digital signature.

3.1.92 Small Modulus: In LBP-PKE, the small modulus p is used for key generation and for modular reduction during decryption.

3.1.93 Statistically Unique: For the generation of n-bit quantities, the probability of two values repeating is less than or equal to the probability of two n-bit random quantities repeating. More formally, an element chosen from a finite set S of n elements is said to be "statistically unique" if the process that governs the selection of this element is such that, for any integer $L \leq n$, the probability that all of the first L selected elements are different is no smaller than the probability of this happening when the elements are drawn uniformly randomly from S.

3.1.94 SVES: Short Vector Encryption Scheme – the encryption scheme defined in this document.

3.1.95 Symmetric Cryptographic Algorithm: A cryptographic algorithm that uses one cryptographic key. Anyone who knows the key can both encrypt and decrypt a message, and can calculate a Message Authentication Code using that key.

3.1.96 Symmetric Cryptography: Methods that allow parties to communicate securely only when they already share some prior secrets, such as the secret key. Contrast: public-key cryptography. See also: secret key.

3.1.97 Symmetric Key: A cryptographic key that is used in symmetric cryptographic algorithms. The same symmetric key that is used for encryption is also used for decryption.

3.1.98 User: A party that uses a public key.

3.1.99 Valid Domain Parameters: a set of domain parameters that satisfies the specific mathematical definition for the set of domain parameters of its family. While a set of mathematical objects may have the general structure of a set of domain parameters, it may not actually satisfy the definition (for example, it may be internally inconsistent) and thus not be valid. See also: domain parameters; public/private key pair; valid key; valid key pair; validation.

3.1.100 Valid Key: a key (public or private) that satisfies the specific mathematical definition for the keys of its family, possibly in the context of its set of domain parameters. While some mathematical objects may have the general structure of keys, they may not actually lie in the appropriate set (for example, they may not lie in the appropriate subgroup of a group or be out of the bounds allowed by the domain parameters) and thus not be valid keys. See also: domain parameters; public/private key pair; valid domain parameters; valid key pair; validation.

3.1.101 Valid Key Pair: a public/private key pair that satisfies the specific mathematical definition for the key pairs of its family, possibly in the context of its set of domain parameters. While a pair of mathematical objects may have the general structure of a key pair, the keys may not actually lie in the appropriate sets (for example, they may not lie in the appropriate subgroup of a group or be out of the bounds allowed by the domain parameters) or may not correspond to each other; such a pair is thus not a valid key pair. See also: domain parameters; public/private key pair; valid domain parameters; valid key; validation.

3.1.102 Validation: See: domain parameter validation; key validation.

3.1.103 Verify: In relation to a Digital Signature means to determine accurately: (1) that the Digital Signature was created during the operational period of a valid Certificate by the private key corresponding to the public-key listed in the Certificate; and (2) the message has not been altered since its Digital Signature was created.

3.2 Acronyms and abbreviations

BS2IP	Bit String to Integer Conversion Primitive
BS2REP	Bit String to Ring Element Conversion Primitive
BS2ROSP	Bit String to Right-padded Octet String Conversion Primitive
BPGM	Blinding Polynomial Generation Method
DP	Decryption Primitive
ES	Encryption Scheme
I2BSP	Integer to Bit String Conversion Primitive
I2OSP	Integer to Octet String Conversion Primitive
IGF	Index Generation Function
IGF-MGF1	Index Generation Function based on Mask Generation Function 1
KGP	Key Generation Primitive
LBP-BPGM1	Blinding polynomial generation method for generating binary blinding polynomials
LBP-DP1	Decryption primitive for use with lattice based polynomial public key decryption
LBP-KGP1	Lattice Based Polynomial Key Generation Primitive
LBP-PKE	Lattice-Based Polynomial Public Key Encryption
MAC	Message authentication code.
MGF	Mask Generation Function
MPM	Message Padding Method
MRGM	Message Representative Generation Method
OS2IP	Octet String to Integer Conversion Primitive
OS2REP	Octet String to Ring Element Conversion Primitive
RE2BSP	Ring Element to Bit String Conversion Primitive
RE2OSP	Ring Element to Octet String Conversion Primitive
ROS2BSP	Right-padded Octet String to Bit String Conversion Primitive
SVDP	Short Vector Decryption Primitive
SVES	Short Vector Encryption Scheme

4 Types of cryptographic techniques

4.1 General model

As stated in Section 1, the purpose of this standard is to provide a reference for specifications of a variety of common public-key cryptographic techniques from which applications may select. Different types of cryptographic techniques can be viewed abstractly according to the following three-level general model.

- *Primitives* – basic mathematical operations. Historically, they were discovered based on number-theoretic hard problems. Primitives are not meant to achieve security just by themselves, but they serve as building blocks for schemes.
- *Schemes* – a collection of related operations combining primitives and additional methods (Section 4.4). Schemes can provide complexity-theoretic security which is enhanced when they are appropriately applied in protocols.
- *Protocols* – sequences of operations to be performed by multiple parties to achieve some security goal. Protocols can achieve desired security for applications if implemented correctly.

From an implementation viewpoint, primitives can be viewed as low-level implementations (e.g., implemented within cryptographic accelerators, or software modules), schemes can be viewed as medium-level implementations (e.g., implemented within cryptographic service libraries), and protocols can be viewed as high-level implementations (e.g., implemented within entire sets of applications).

This standard contains only specifications of schemes.

4.2 Schemes

The following types of schemes are defined in this standard:

- Encryption Schemes (ES), in which any party can encrypt a message using a recipient's public key, and only the recipient can decrypt the message by using its corresponding private key. Encryption schemes may be used for establishing secret keys to be used in symmetric cryptography.

Schemes in this standard are presented in a general form based on certain primitives and additional methods. For example, the encryption scheme defined in this standard is based on a key generation primitive, a decryption primitive, and a blinding polynomial generation method.

Schemes also include key management operations, such as selecting a private key or obtaining another party's public key. For proper security, a party needs to be assured of the true owners of the keys and domain parameters and of their validity. Generation of domain parameters and keys needs to be performed properly, and in some cases validation also needs to be performed. While outside the scope of this standard, proper key management is essential for security.

An *Encryption Scheme* is specified by providing the following:

- Name
- Type (e.g. Asymmetric Public-key Encryption Scheme)
- Options (Key Type, Primitives, Parameters)
- Operations
  - Key Pair Generation
  - Key Pair Validation
  - Public Key Validation
  - Encryption Operation
    - Input

    - Output
- Decryption Operation
    - Input
    - Output

An encryption scheme specification may also include the following:

- Security Considerations
- Implementation Considerations
- Related Standards

The specifications are functional specifications, not interface specifications. As such, the format of inputs and outputs and the procedure by which an implementation of a scheme is invoked are outside the scope of this standard. See Annex E for more information on input and output formats.

4.3 Additional methods

This standard specifies the following additional methods:

- Blinding Polynomial Generation Methods, which are components of encryption schemes.
- Auxiliary Functions, which are building blocks for other additional methods.
    - Index generation functions
    - Mask Generation Functions
    - Hash Functions, which are used as the core of Index generation functions and of Mask Generation Functions.

The specified additional methods are required for conformant use of the schemes. The use of an inadequate message encoding method, key derivation function, or auxiliary function may compromise the security of the scheme in which it is used. Therefore, any implementation which chooses not to follow the recommended additional methods for a particular scheme should perform its own thorough security analysis of the resulting scheme.

4.4 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 IGF, 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 shall 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 consistently 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.

5 Mathematical notation

When referring to mathematical objects and data objects in this standard, the following notation is used. Throughout the document, numbers at the end of variable names are used to distinguish different, but related values (e.g. *df1*, *df2*, *df3* or *Dmin1*, *Dmin2*, etc.).

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
$\times$	Indicates the convolution product of two polynomials and is also used to indicate multiplication of integers
$\oplus$ or XOR	Exclusive OR function
$\|\|$	Concatenation. $A\|\|B$ is the 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 $A\|\|B$.
:=	Initialization. $a := b$ means initialize or set the value of a equal to the value of b.
A	Lower-bound decryption coefficient, used in decryption process to reduce into correct interval
ceil[.] or $\lceil . \rceil$	Ceiling function (i.e. the smallest integer greater than or equal to the contents of [.])
db	The number of random bits used as input for encryption
d_f	An integer specifying the number of ones in the polynomials that comprise the private key value f (also specified as $df1$, $df2$, and $df3$, or as dF)
d_g	An integer specifying the number of ones in the polynomials that comprise the temporary polynomial g (often specified as dG)
d_r	An integer specifying the number of ones in the blinding polynomial r in SVES. (also specified as $dr1$, $dr2$, and $dr3$)
e	Encrypted message representative, a polynomial, computed by an encryption primitive
E	Encrypted message, an octet string.
ES	(Asymmetric) encryption scheme.
f	Private key in SVES.
F	In SVES, a polynomial that is used to calculate the value f when $f=1+pF$.
floor[.] or $\lfloor . \rfloor$	Floor function (i.e. the largest integer less than or equal to the contents of [.])

g	In SVES, a temporary polynomial used in the key generation process.
GCD(*a*, *b*)	Greatest Common Divisor of two non-negative integers *a* and *b*.
h	Public key
Hash( )	A cryptographic hash function computed on the contents of ( )
hLen	Length in octets of a hash value.
i	An integer
k	Security level in bits.
m	The message, an octet string, which is encrypted in SVES.
M	In SVES, the padded and formatted message representative octet string used during encryption and decryption.
m'	The message representative polynomial which is submitted to the encryption primitive in the SVES encryption scheme.
mod *q*	Used to reduce the coefficients of a polynomial into some interval of length *q*
mod *p*	Used to reduce a polynomial to an element of the polynomial ring mod *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
r	In LBP-PKE, the encryption blinding polynomial (generated from the hash of the padded message *M* in SVES)
x	The integer input to or output from integer conversion primitives
X	The indeterminate used in polynomials
$\boldsymbol{Z}$	The ring of integers
$\boldsymbol{Z}_q$	The ring of integers mod *q*.

6 Polynomial representation and operations

6.1 Introduction

The cryptographic techniques specified in this standard require arithmetic in quotient polynomial rings, also called convolution polynomial rings. 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.

6.2 Polynomial representation

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 are simply denoted by 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 $a(X) = a_0 + a_1X + a_2X^2 + a_3X^3 + \ldots + a_iX^i + \ldots + a_{N-1}X^{N-1}$, unless otherwise specified. Polynomial operations

6.3 Polynomial operations

6.3.1 Polynomial generation

Generates polynomials by creating for each polynomial, a list of the indices of the +1 coefficients followed by a list of the indices of the -1 coefficients.

If a single polynomial is generated (non-product form), indices_counts contains a single value of the total number of indices (for +1 and -1 coefficients combined).

If multiple polynomials are generated (for product form), their lists of indices are sequentially stored in the indices buffer. Each byte of indices_counts contains the total number of indices (for +1 and -1 coefficients combined) for a single polynomial, beginning with the low-order byte for the first polynomial. The high-order byte is unused.

6.3.1.1 Pseudo random trinary polynomial generator

A pseudo random trinary polynomial generator is instantiated with parameters N and d_f, an index generation function (IGF) with a seed *seed*, a the hash function Hash() chosen to parameterize IGF(), and the minimum number of hash calls for the IGF to make, *minCalls*.

Input: N, d_f, *seed*, Hash(), *minCalls*.

Output: A pseudo random trinary polynomial f.

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

a) Call the IGF with hash function Hash() and input *seed*, N, *minCalls* to obtain the IGF state s.
b) Set $f := 0$
c) Set $t := 0$
d) While $t < d_f$ do
    1) Call the IGF with input s to obtain an integer i mod N.
    2) If $f_i = 0$
        i) Set $f_i := 1$
        ii) Set $t := t + 1$
e) Set $t := 0$
f) While $t < d_f$ do
    1) Call the IGF with input s to obtain an integer i mod N and the updated state s. If the IGF outputs "error", output "error".
    2) If $f_i = 0$
        i) Set $f_i := -1$
        ii) Set $t := t + 1$
g) Return f

6.3.2 Polynomial multiplication

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

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

All multiplications of polynomials *a* and *b*, represented as $a \times b$, are taken to occur in the ring **Z**[*X*]/(X^N – 1) unless otherwise noted.

6.3.2.1 Product form polynomial multiplication over the ring

Let **Z** be the ring of integers. The polynomial ring over **Z**, denoted **Z**[*X*], is the set of all polynomials with coefficients in the integers. The *convolution polynomial ring (over* **Z***) of degree N* is the quotient ring **Z**[*X*]/(X^N – 1). The product *c* of two polynomials *a*, *b* ε **Z**[*X*]/(X^N – 1) can be computed efficiently using the product form polynomial multiplication when *a* is a polynomial form $a = a_1 \times a_2 + a_3$. The procedure is as follows:

a) $t_1 = a_1 \times b$
b) $t_2 = a_2 \times t_1$
c) $t_3 = a_3 \times b$
d) $c = t_2 + t_3$

The additions + and multiplications × here are polynomial operations over the ring, as specified in the previous section. The product form polynomial multiplication method is used in the product form method in SVES.

6.3.3 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.

6.3.4 Inversion in (Z/*q*Z)[*X*]/(X^N – 1)

For certain cryptographic operations such as key generation, it is necessary to take the inverse of a polynomial in (**Z**/*q***Z**)[*X*]/(X^N – 1). This section describes the algorithms necessary for inversion in this ring.

6.3.4.1 The Polynomial Division Algorithm in $Z_p[X]$

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*-1 whose leading coefficient b_N is not 0.

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

Operation: Polynomial Division Algorithm in $Z_p[X]$ shall be computed by the following or an equivalent sequence of steps;

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

6.3.4.2 The Extended Euclidean Algorithm in $Z_p[X]$

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 $a \times u + b \times v = d$. 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 $d = \text{GCD}(a, b)$ and $a \times u + b \times v = d$.

Operation: Extended Euclidean Algorithm in $Z_p[X]$ shall be computed by the following or an equivalent sequence of steps;

a) If $b = 0$ then return $(1, 0, a)$
b) Set $u := 1$
c) Set $d := a$
d) Set $v_1 := 0$
e) Set $v_3 := b$
f) While $v_3 \neq 0$ do
    1) Use the division algorithm (6.3.4.1) to write $d = v_3 \times q + t_3$ with deg $t_3 <$ deg v_3
    2) Set $t_1 := u - q \times v_1$
    3) Set $u := v_1$
    4) Set $d := v_3$
    5) Set $v_1 := t_1$
    6) Set $v_3 := t_3$
g) Set $v := (d - a \times u)/b$ [This division is exact, i.e., the remainder is 0]
h) Return (u, v, d)

6.3.4.3 Inverses in $Z_p[X]/(X^N - 1)$

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 $\text{GCD}(a, X^N - 1)$ 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 $a \times b = 1$ in $\mathbf{Z}_p[X]/(X^N - 1)$ if a is invertible in $\mathbf{Z}_p[X]/(X^N - 1)$, otherwise FALSE.

Operation: Inverses in $Z_p[X]/(X^N - 1)$ shall be computed by the following or an equivalent sequence of steps;

a) Run the Extended Euclidean Algorithm (6.3.4.2) with input a and $(X^N - 1)$. Let (u, v, d) be the output, such that $a \times u + (X^N - 1) \times v = d = \text{GCD}(a, (X^N - 1))$.
b) If deg $d = 0$
c) Return $b = d^{-1} \pmod{p} \times u$
d) Else return FALSE

6.3.4.4 Inverses in $Z_{p^r}[X]/(X^N - 1)$

For key generation in this standard it is necessary to calculate inverses in $Z_q[X]/(X^N - 1)$, where q is a power of 2. In this case, the Inversion Algorithm (6.3.4.3) may be used to find the inverse of $a(X)$ in the quotient ring $(R/2R)[X]/(M(X))$. Then the following algorithm may be used to lift it to an inverse of $a(X)$ in the quotient ring $(R/p^eR)[X]/(M(X))$ with higher powers of the prime 2 (or any prime p).

Input. A prime p in a Euclidean ring R, a monic polynomial $M(X) \varepsilon R[X]$, a polynomial $a(X) \varepsilon R[X]$, and an exponent e.

Output. An inverse $b(X)$ of $a(X)$ in the ring $(R/p^eR)[X]/(M(X))$ if the inverse exists, otherwise FALSE.

a) Use the Inversion Algorithm 6.3.4.4 to compute a polynomial $b(X) \varepsilon R[X]$ that gives an inverse of $a(X)$ in $(R/pR)[X]/(M(X))$. Return FALSE if the inverse does not exist. [The Inversion Algorithm may be applied here because R/pR is a field, and so $(R/pR)[X]$ is a Euclidean ring.]
b) Set $n \leftarrow 2$
c) While $e > 0$ do
d) $b(X) \leftarrow 2 \times b(X) - a(X) \times b(X)^2 \pmod{M(X)}$, with coefficients computed modulo p^n
e) Set e $\leftarrow \lfloor e/2 \rfloor$
f) Set $n \leftarrow 2 \times n$
g) Return $b(X)$ mod $M(X)$ with coefficients computed modulo p^e.

7 Data Types and Conversions

7.1 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.

When a bit string or a octet string is being encoded into a polynomial with coefficients reduced mod q (a "ring element"), where q is usually either 128 or 256, the integer coefficients are mapped individually to bit or octet strings, which are then concatenated. This mapping and its reverse are described in the conversion primitives OS2REP, BS2REP, RE2OSP and RE2BSP in 7.5 and 7.5.

This standard does not specify a single algorithm for converting from bit/octet strings to ternary polynomials in an unbiased and reversible fashion. Instead, the standard uses two algorithms, which are

defined inline in the techniques that use them. One algorithm is reversible but biased; the other is unbiased but non-reversible.

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

7.2.1 Integer to Bit String Primitive (I2BSP)

I2OSP converts a nonnegative integer to a bit string of a specified length.

Input: *i*, nonnegative integer to be converted; *bLen*, intended length of the resulting bit string

Output: *B*, corresponding bit string of length *bLen*

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

a) If $x \geq 2^{xLen}$, output "integer too large" and stop.

b) Write the integer x in its unique $xLen$-bit representation in base 2:
$x = x_{xLen-1} \times 2^{xLen-1} + x_{xLen-2} \times 2^{xLen-2} + \ldots + x_1 \times 2 + x_0$
where $x_i = 0$ or 1 (note that one or more leading bits will be zero if x is less than 2^{xLen-1}).

c) Output the bit string $x_{xLen-1}\, x_{xLen-2} \ldots x_1\, x_0$.

7.2.2 Bit String to Integer Primitive (BS2IP)

BS2IP converts a bit string to a nonnegative integer.

Input: *B*, bit string to be converted (*bLen* is used to denote the length of *B*)

Output: *x*, corresponding nonnegative integer

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

a) If *B* is of length 0, output 0.

b) Let $b_{bLen-1}\, b_{bLen-2} \ldots b_1\, b_0$ be the bits of *B* from leftmost to rightmost.

c) Let $x = b_{bLen-1} \times 2^{bLen-1} + b_{bLen-2} \times 2^{bLen-2} + \ldots + b_1 \times 2 + b_0$.

d) Output *x*.

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

7.3.1 Integer to Octet String Primitive (I2OSP)

I2OSP converts a nonnegative integer to an octet string of a specified length.

Input: *x*, nonnegative integer to be converted; *oLen*, intended length of the resulting octet string

Output: *O*, corresponding octet string of length *oLen*

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

a) If $x \geq 256^{oLen}$, output "integer too large" and stop.

b) Write the integer x in its unique $oLen$-digit representation in base 256:
$x = o_{oLen-1} \times 256^{oLen-1} + o_{oLen-2} \times 256^{oLen-2} + \ldots + o_1 \times 256 + o_0$
where $0 \leq o_i < 256$ (note that one or more leading digits will be zero if o is less than 256^{oLen-1}).

c) For for $1 \le x \le oLen$, let the octet O_i be the concatenation of the bits in the integer representation of o_{oLen-i} , where left-most bit of the octet is the high order bit of the binary representation. Output the octet string

$O = O_1 O_2 \ldots O_{oLen}$.

NOTE—As an example, the integer 944 has the three-digit representation $944 = 0 \times 256^2 + 3 \times 256 + 178$. The corresponding octet string, expressed in integer values, is 0 3 178; as binary values, it is

00000000 00000011 10110010

and in hexadecimal it is 00 03 b2.

7.3.2 Octet String to Integer Primitive (OS2IP)

OS2IP converts an octet string to a nonnegative integer.

Input: *x*, nonnegative integer to be converted; *oLen*, intended length of the resulting octet string

Output: *O*, corresponding octet string of length *oLen*

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

a) If *O* is of length 0, output 0.

b) Let $O_1 O_2 \ldots O_{oLen}$ be the octets of *O* from first to last, and let o_{oLen-j} be the integer value of the octet O_j for $1 \le j \le oLen$, where the integer value is represented as an octet (x.e., an eight-bit string) most significant bit first.

c) Output $x = o_{oLen-1} \times 256^{oLen-1} + o_{oLen-2} \times 256^{oLen-2} + \ldots + o_1 \times 256 + o_0$.

7.4 Converting Between Bit Strings and Right-Padded Octet Strings (BS2ROSP and ROS2BSP)

This section gives the primitives used to convert between bit strings and right-padded octet strings.

7.4.1 Bit String to Right-Padded Octet String Primitive (BS2ROSP)

Input: *B*: bit string to be converted; *oLen*: intended length of the resulting octet string

Output: *O*, corresponding octet string of length *oLen*

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

a) Set *bLen* equal to the length of *x* in bits.

b) If $bLen > 8 \times oLen$, output "input too long" and stop.

c) Append $(8 \times oLen - bLen)$ zero bits to the end of *x*.

d) Let $b_0 b_1 \ldots b_{xLen-2} b_{xLen-1}$ be the bits of *B* from first to last. For $0 \le i < oLen - 1$, let the octet $O_i = b_{8i} b_{8i+1} \ldots b_{8i+7}$. Output the octet string

$O = O_0 O_1 \ldots O_{oLen-1}$.

7.4.2 Right-Padded Octet String to Bit String Primitive (ROS2BSP)

ROS2BSP converts an octet string to a bit string of a specified length.

Input: *O*: octet string to be converted; *bLen*: intended length of the resulting bit string

Output: *B*: corresponding bit string of length *bLen*

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

a) Set *oLen* equal to the length of *O* in octets.
b) If $bLen > 8 \times oLen$, output "input too short" and stop.
c) For $0 \le i < oLen - 1$, consider the octet O_i to be the bits $b_{8i}\ b_{8i+1} \ldots b_{8i+7}$.
d) If any of the bits $b_{bLen-1} \ldots b_{8 \times oLen - 1}$ are non-zero, output "non-zero bits found after end of bit string" and stop.
e) Output the bit string
$B = b_0\ b_1 \ldots b_{bLen-1}$.

7.5 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. This section provides the appropriate conversion primitives.

7.5.1 Ring Element to Bit String Primitive (RE2BSP)

RE2BSP converts a ring element to a bit string.

Input: *a*: ring element to be converted, equal to $a_0 + a_1 X + a_2 X^2 + \ldots + a_{N-1} X^{N-1}$; *N*: dimension of ring; *q*: larger modulus: all coefficients of the ring element are between 0 and *q*-1.

Output: *B*: resulting bit string.

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

a) For $j = 0$ to $N-1$:
b) Set A_j equal to the smallest positive representation of $a_j \bmod q$.
c) Set B_j = I2BSP (A_j, ceil[$\log_2 q$]). If any of the calls to I2BSP output an error, output that error and stop.
d) Output the bit string $B = B_0\ B_1 \ldots B_{N-1}$.

NOTE—As an 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 BS2ROSP, 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).

7.5.2 Bit String to Ring Element Primitive (BS2REP)

BS2REP converts a bit string to a ring element.

Input: *B*: bit string to be converted; *N*: dimension of ring; *q*: larger modulus: all coefficients of the ring element are between 0 and *q*-1.

Output: *a*: resulting ring element, equal to $a_0 + a_1 X + a_2 X^2 + \ldots + a_{N-1} X^{N-1}$

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

a) If the length of *B* is not equal to $N \times$ ceil[$\log_2 q$], output "bit string incorrect length" and stop.
b) Consider *B* to be the series of bit strings $B = B_0\ B_1 \ldots B_{N-1}$., where each B_j is of length ceil[$\log_2 q$] bits.

c) For $j = 0$ to N-1, set a_j = BS2IP (B_j). If BS2IP outputs an error, output "error".

d) Output $a = a_0 + a_1 X + a_2 X^2 + \ldots + a_{N-1} X^{N-1}$.

7.6 Converting Between Ring Elements and (Compact) Octet Strings (RE2OSP and OS2REP)

This section gives the primitives for converting between ring elements and octet strings using the minimum number of bits per coefficient.

7.6.1 Ring Element to Octet String Primitive (RE2OSP)

RE2OSP converts a ring element to an octet string.

Input: a: ring element to be converted, equal to $a_0 + a_1 X + a_2 X^2 + \ldots + a_{N-1} X^{N-1}$; N: dimension of ring; q: larger modulus to be passed to RE2BSP: all coefficients of the ring element are between 0 and q-1

Output: O: corresponding octet string

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

a) Convert the ring element a to a bit string bA using RE2BSP.

b) Convert the bit string bA to the octet string O using BS2ROSP.

c) Output O.

7.6.2 Octet String to Ring Element Primitive (OS2REP)

OS2REP converts an octet string to a ring element.

Input: O: octet string to be converted; N: dimension of ring; q: larger modulus: all coefficients of the ring element are between 0 and q-1.

Output: a: resulting ring element, equal to $a_0 + a_1 X + a_2 X^2 + \ldots + a_{N-1} X^{N-1}$

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

a) If the length of O is not equal to $N \times \text{ceil}[\log_{256} q]$, output "octet string incorrect length" and stop.

b) Convert the octet sting O to the bit string bA using ROS2BSP.

c) Convert the bit string bA to the ring element a using BS2REP.

d) Output a.

7.7 Converting Between Ring Elements and Padded Octet Strings (RE2POSP and POS2REP)

This section gives the primitives for converting between ring elements and octet strings such that coefficients of the ring element do not cross octet boundaries.

7.7.1 Ring Element to Padded Octet String Primitive (RE2POSP)

RE2OSP converts a ring element to an octet string.

Input: a: ring element to be converted, equal to $a_0 + a_1 X + a_2 X^2 + \ldots + a_{N-1} X^{N-1}$; N: dimension of ring; q: larger modulus to be passed to RE2BSP: all coefficients of the ring element are between 0 and q-1

Output: O: corresponding octet string

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

a) For $j = 0$ to N-1:
b) Set A_j equal to the smallest positive representation of a_j mod q.
c) Set O_j = I2OSP (A_j, ceil[$\log_{256} q$]). If any of the calls to I2OSP output an error, output that error and stop.
d) Output the octet string $O = O_0 O_1 \ldots O_{N-1}$.

7.7.2 Octet String to Ring Element Primitive (OS2REP)

OS2REP converts an octet string to a ring element.

Input: O: octet string to be converted; N: dimension of ring; q: larger modulus: all coefficients of the ring element are between 0 and q-1.

Output: a: resulting ring element, equal to $a_0 + a_1 X + a_2 X^2 + \ldots + a_{N-1} X^{N-1}$

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

a) If the length of O is not equal to $N \times$ ceil[$\log_{256} q$], output "octet string incorrect length" and stop.
b) Consider O to be the series of octet strings $O = O_0 O_1 \ldots O_{N-1}$., where each O_j is of length ceil[$\log_{256} q$] bits.
c) For $j = 0$ to N-1, set a_j = OS2IP (O_j). If OS2IP outputs an error, output "error".
d) Output $a = a_0 + a_1 X + a_2 X^2 + \ldots + a_{N-1} X^{N-1}$.

7.8 Converting Between Trinary Ring Elements and Octet Strings (TRE2OSP and OS2TREP)

This section gives the primitives for converting between ring elements and octet strings such that coefficients of the ring element do not cross octet boundaries.

7.8.1 Trinary Ring Element to Octet String Primitive (TRE2OSP)

RE2OSP converts a trinary ring element to an octet string.

Input: a: ring element to be converted, equal to $a_0 + a_1 X + a_2 X^2 + \ldots + a_{N-1} X^{N-1}$, where the coefficients a_i are all in the set {-1, 0, 1}; N: dimension of ring

Output: O: corresponding octet string

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

a) Set $k = 0$
b) For $j = 0$ to N-1:
    1) If $a_j = 1$
        i) Set O_k = I2OSP (j, ceil[$\log_{256} qN$])
        ii) Set $k = k+1$
c) For $j = 0$ to N-1:
    1) If $a_j = -1$
        i) Set O_k = I2OSP (j, ceil[$\log_{256} qN$])
        ii) Set $k = k+1$
d) Output the octet string $O = O_0 O_1 \ldots O_{N-1}$.

7.8.2 Octet String to Trinary Ring Element Primitive (OS2TREP)

OS2REP converts an octet string to a ring element whose coefficients are all in the set {-1, 0, 1}.

Input: *O*: octet string to be converted; *N*: dimension of ring; d_+, expected number of +1s; d_-, expected number of -1s.

Output: *a*: resulting ring element, equal to $a_0 + a_1 X + a_2 X^2 + \ldots + a_{N-1} X^{N-1}$

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

a) If the length of *O* is not equal to $(d_+ + d_-) \times \text{ceil}[\log_{256} N]$, output "octet string incorrect length" and stop.
b) Consider *O* to be the series of octet strings $O = O_0 O_1 \ldots O_{(d+ + d-)}$., where each O_j is of length $\text{ceil}[\log_{256} N]$ bits.
c) For $k = 0$ to d_+:
    1) If k != 0
        i) If OS2IP (O_k) <= OS2IP (O_{k-1}), output "error".
    2) Set $a_{\text{OS2IP }(Ok)} = 1$. If OS2IP outputs an error, output "error".
d) For $k = d_+ + 1$ to $d_+ + d_-$:
    1) If k != $d_+ + 1$
        i) If OS2IP (O_k) <= OS2IP (O_{k-1}), output "error".
    2) Set $a_{\text{OS2IP }(Ok)} = 1$. If OS2IP outputs an error, output "error".
e) Output $a = a_0 + a_1 X + a_2 X^2 + \ldots + a_{N-1} X^{N-1}$.

8 Supporting algorithms

8.1 Overview

In order to perform the 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 described in FIPS 180 is typically chosen). This section defines the algorithms that shall be used to meet this standard.

8.2 Hash Functions

Hash functions are used in two distinct situations in this standard: as the core of a mask generation function, and as the core of a pseudo-random bit generator. For security purposes, the hash function should be chosen at a strength commensurate to the desired security level. The recommended parameter sets in this document specify hash functions appropriate to their security levels.

The only currently supported hash functions for use within this standard are SHA-1 and SHA-256 (see FIPS 180).

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 ROS2BSP and BS2ROSP (sections 7.4.1 and 7.4.2) may be used.

8.3 Encoding Methods

8.3.1 General

Before a message is encrypted, it shall be processed to provide certain desirable security properties such as semantic security. In this section, the auxiliary methods for manipulating data for the encryption scheme are listed. These currently consist of specific methods for generating the blinding polynomial *r*.

8.3.2 Blinding Polynomial Generation Methods (BPGM)

8.3.2.1 General

In order to provide plaintext awareness, a blinding polynomial generation method (BPGM) shall be used to generate a blinding polynomial *r* from the padded message *pm*. This section contains two BPGMs. The first utilizes the standard polynomial convolution method, and the second utilizes the optimized polynomial convolution method.

8.3.2.2 lbp-bpgm-3

The blinding polynomial *r* shall be generated deterministically from the message *m* and the random value *b* using a pseudo-random number generator.

Components: The parameters *N* and d_r (or dr_1, dr_2, dr_3), and a Boolean value *is_product_form*, the chosen index generation function IGF(), the hash function Hash() chosen to parameterize IGF(), the polynomial index generation constant *c*, and the minimum number of hash calls for the IGF to make, *minCallsR*.

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

Output: The blinding polynomial, which is a polynomial *r*.

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

a) Call the IGF with hash function Hash() and input *seed*, *N*, *c*, *minCallsR* to obtain the IGF state *s*.
b) If *is_product_form* = false:
    1) Generate *r* using pseudo random trinary polynomial generator with parameter *N* and d_r, using IGF with state *s*.
c) If *is_product_form* = true:
    1) Generate three polynomials r_1, r_2, r_3 using pseudo random trinary polynomial generator with parameter *N* and (dr_1, dr_2, dr_3), using IGF with state *s*.
    2) Compute $r = r_1 \times r_2 + r_3$.
d) Return *r*

8.4 Supporting Algorithms

In order to perform the 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 (see FIPS 180) is typically chosen). This section defines the algorithms that shall be used to meet this standard.

8.4.1 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 and SHA-256 (see FIPS 180).

This standard only permits the use of one mask generation function, MGF-TP-1. This function takes as input an octet string and the desired degree of the output, and produces a ternary polynomial of the appropriate degree. The only hash functions supported for use with this mask generation function are SHA-1 and SHA-256 (see FIPS 180).

8.4.1.1 Mask Generation Function for Ternary Polynomials (MGF-TP-1)

Components: A hash function *Hash* with output length *hLen* octets.

Input: an octet string *seed* of length *seedLen* octets; the degree *N*, an integer; an argument *hashSeed*, taking the values "yes" or "no"; and the minimum number of calls *minCallsMask*, an integer

Output: A polynomial *i* of degree *N*-1; or “error”.

Operation: The integer and state shall be produced by the following or an equivalent sequence of steps:

a) If *seedLen*+4 exceeds any input length limitation on the hash function *Hash*, output “error” and exit
b) If *minCallsMask* exceeds 2^{32}, output “error” and exit.
c) Check the value of *hashSeed*.
    1) If *hashSeed* = "yes", set the octet string *Z* to *Hash*(*seed*) and the integer *zLen* to *hLen*.
    2) If *hashSeed* = "no", set the octet string *Z* to *seed* and the integer *zLen* to *seedLen*.
d) Initialize the octet string *buf* to be a zero-length octet string.
e) Initialize *counter*:= 0.
f) Initialize *N* and *c* with the provided values. Set *cLen* = ceil (*c*/8).
g) While *counter* < *minCallsR* 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 *buf* = *buf* || *H*.
    4) Increment *counter* by one.
h) Initialize *i* to be the null polynomial and *cur*, a pointer to the current coefficient of *i*, to be 0.
i) For each octet *o* in *buf:*
    1) Convert *o* to an integer *O* using OS2IP.
    2) If *O* >= 243 (= 3^5) discard *O*, move to the next octet, and go to step d)1).
    3) Set i_{cur} = *O* mod 3; if *cur* = *N* output *i*; set *cur* = *cur* + 1; set *O* = (*O* – *O* mod 3) / 3.
    4) Set i_{cur} = *O* mod 3; if *cur* = *N* output *i*; set *cur* = *cur* + 1; set *O* = (*O* – *O* mod 3) / 3.
    5) Set i_{cur} = *O* mod 3; if *cur* = *N* output *i*; set *cur* = *cur* + 1; set *O* = (*O* – *O* mod 3) / 3.
    6) Set i_{cur} = *O* mod 3; if *cur* = *N* output *i*; set *cur* = *cur* + 1; set *O* = (*O* – *O* mod 3) / 3.
    7) Set i_{cur} = *O*; if *cur* = *N* output *i*; set *cur* = *cur* + 1
j) If *cur* < *N*:
    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 *buf* = *H*.
4) Increment *counter* by one.
5) Return to step i).

k) Output *i*.

8.4.2 Index generation function

The term "index generation function", as used in this standard, applies to functions which are initialized with a seed in the form of an octet string and may then be called repeatedly, producing an integer in a specified range on each call.

An IGF may be deterministic or non-deterministic. A deterministic IGF is parameterized by a hash function; the only hash functions supported for use with the IGFs in this standard are SHA-1 and SHA-256 (see FIPS 180). On initialization, it takes as input a *seed*, which is an octet string; a modulus *N*; an index generation constant *c*; and the desired minimum number of calls to the underlying hash function, *minCallsR*. It outputs an integer in the range [0, *N*-1] and the internal state *s*. On subsequent calls, it takes as input the current state *s* and outputs an octet string of length *oLen* and the updated internal state *s*.

This standard permits the use of a deterministic index generation function based on a hash function and a nondeterministic index generation function based on a random bit generator.

8.4.2.1 Index generation function (IGF-2)

Components: A hash function *Hash* with output length *hLen* octets.

Input:

EITHER: an octet string *seed* of length *seedLen* octets; the modulus *N*, an integer; an argument *hashSeed*, taking the values "yes" or "no"; the index generation constant *c*, an integer; and the minimum number of calls *minCallsR*, an integer

OR: the state *s*.

Output: An integer *i* and the state *s*; or "error".

Operation: The integer and state shall be produced by the following or an equivalent sequence of steps:

a) If *s* is not provided:
1) If *seedLen*+4 exceeds any input length limitation on the hash function *Hash*, output "error" and exit
2) If *minCallsR* exceeds 2^{32}, output "error" and exit.
3) Check the value of *hashSeed*.
i) If *hashSeed* = "yes", set the octet string *Z* to *Hash*(*seed*) and the integer *zLen* to *hLen*.
ii) If *hashSeed* = "no", set the octet string *Z* to *seed* and the integer *zLen* to *seedLen*.
4) Initialize *totLen* to 0. Intialize *remLen* to 0.
5) Initialize the bit string buf to be a zero-length bit string.
6) Initialize *counter*:= 0.
7) Initialize *N* and *c* with the provided values.
8) While *counter* < *minCallsR* do
i) Convert *counter* to an octet string *C* of length 4 octets using I2OSP.

ii) Compute *Hash*(Z || C) with the selected hash function to produce an octet string *H* of length *hLen* octets.

iii) Let *buf* = *buf* || OS2BSP(*H*).

iv) Increment *counter* by one.

9) Set *remLen* = *totLen* = *minCallsR* × 8 × *hLen*.

b) Otherwise (if *s* is provided):

1) Extract the values *Z, totLen, remLen, buf, counter, N, c* from the state *s*. (The details of how they are stored in *s* may be determined by the implementer).

c) Set *totLen*:=*totLen* + *c*.

d) If *totLen* exceeds $hLen \times 8 \times 2^{32}$, output "error" and exit.

e) If *remLen* < c

1) Let the bit string *M* be the trailing *remLen* bits in *buf*.

1) Let *tmpLen*:=*c* – *remLen*.

2) Let *cThreshold* = *counter* + ceil[*tmpLen*/*hLen*].

3) While *counter* < *cThreshold* do

i) Convert *counter* to an octet string *C* of length 4 octets using I2OSP.

ii) Compute *Hash*(Z || C) with the selected hash function to produce an octet string *H* of length *hLen* octets.

iii) Let *M* = *M* || OS2BSP(*H*).

iv) Increment *counter* by one. If *tmpLen* > 8 × *hLen*, decrement *tmpLen* by 8 × *hLen*.

4) Set *remLen*:=8 × *hLen* – *tmpLen*. Set *buf*:=*H*.

f) else

1) Set *M* equal to the trailing *remLen* bits of *buf*.

2) Set *remLen*:=*remLen* – *c*.

g) Set the bit string *b* to the leading *c* bits in *M*,

h) Convert *b* to an integer *i* using OS2IP.

i) If $i \geq 2^c - (2^c \bmod N)$ go back to step 3.

j) Store the values *Z, totLen, remLen, counter, N, cLen* and *c* in the state *s*. (The details of how they are stored in *s* may be determined by the implementer).

k) Output *i* mod *N* and *s*.

8.4.2.2 Index generation function (IGF-RBG)

This IGF is based on any approved random bit generator

Components: An approved random bit generator RBG

Input: The modulus *N*, an integer; the index generation constant *c*, an integer.

Output: An integer *i*

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

a) Set *cLen* = ceil (c/8).

b) Obtain a bit string *b* of length 8 × *cLen* bits from RBG.

c) Convert *b* to an octet string *o* using BS2OSP.

d) Set the leftmost 8*cLen* - *c* bits of *o* to 0.

e) Convert *o* to an integer *i* using OS2IP.

f) If $i >= 2^c - (2^c \bmod N)$ go back to step 3.

g) Output $i \bmod N$.

9 Short Vector Encryption Scheme (SVES)

The following section defines the supported encryption schemes. The only encryption scheme currently supported is SVES. SVES stands for Short Vector Encryption Scheme (see for more information).

9.1 Encryption Scheme (SVES) Overview

The general encryption scheme is a sequence of operations that are performed based on the choices of the parameters, primitives, encoding functions and supporting algorithms. In order to perform all of the SVES encryption scheme operations, all of the Components shall be specified.

9.2 Encryption Scheme (SVES) Operations

The SVES encryption scheme consists of the five operations key generation, key pair validation, public key validation, encryption and decryption. These operations are defined generally in this section without assuming any specific choices of the Components listed in Section 9.1 Encryption Scheme (SVES) Overview.

9.2.1 Key Generation

A key pair shall be generated using the following or a mathematically equivalent set of steps. Note that the algorithm below outputs only the values f and h. In some applications it may be desirable to store the values f^{-1} and g as well. This standard does not specify the output format for the key as long as it is unambiguous

Components: The parameters N, q, p, dF (or df_1, df_2 and df_3), d_g, *is_product_form*; EITHER an Approved random number generator capable of generating unbiased output in the range (0, N-1) OR an index generation function IGF that takes an approved random bit generator RBG and the polynomial index generation constant c used by the IGF.

Input: None

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

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

a) Set the polynomial $F := 0$.

b) If *is_product_form* = False:

   1) Call pseudo random trinary polynomial generator with input N, c, dF and RBG to obtain a trinary polynomial F.

c) If *is_product_form* = True:

   1) Call pseudo random trinary polynomial generator with input N, c, df_1, df_2, df_3 and RBG to obtain a trinary polynomials F_1, F_2 and F_3.

   2) Set $F = F_1 \times F_2 + F_3$

d) Compute the polynomial $f := 1 + p \times F$ in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$

e) Compute the polynomial f^{-1} (i.e. the polynomial f^{-1} such that $f^{-1} \times f = f \times f^{-1} = 1$) in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$. If f^{-1} does not exist, go to step a).

f) Call pseudo random trinary polynomial generator with input N, c, d_g and RBG to obtain a trinary polynomial g.

g) Set $g = g+1$

h) Check that g is invertible mod q. If it is not, go back to step f).

i) Compute the polynomial $h := f^{-1} \times g \times p$ in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$

j) If *is_product_form* = False:
   1) Output *f*, *h*
k) If *is_product_form* = True:
   1) Output *f*, *h*, F_1, F_2, F_3

9.2.2 Encryption Operation

This section defines the Encryption operation. Note that within the definition of the spaces may be definitions of additional variables (e.g. when defining D_r, the values *dr*1, *dr*2 and *dr*3 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 shall be a multiple of 8.
— The chosen Mask Generation Function and associated parameters.
— The chosen Blinding Polynomial Generation Method and the associated parameters
— The OID, an octet string
— The number of bits of public key to hash, *pkLen*.
— The minimum message representative weight, d_{m0}.
— The minimum number of calls to generate the masking polynomial, *minCallsMask*.
— The maximum message length *maxMsgLenBytes*
— The minimum number of calls to generate the blinding polynomial, *minCallsR*.
— The length of the encoding buffer, *bufferLenBits*

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:

a) Calculate *octL* = the *lLen*-octet-long encoding of the message length *l*.
b) If *l* > *maxLen*, output "message too long" and stop.
c) Randomly select an octet string *b* of length *bLen* using a random number generator with at least 8 × *bLen* bits of entropy content.
d) Form the octet string *p0*, consisting of the 0 byte repeated (*maxMsgLenBytes* + 1 - *l*) times.
e) Form the octet string *M* of length *bufferLenBits*/8 as *b* || *octL* || *m* || *p0*.
f) Convert *M* to a bit string *Mbin* using OS2BSP.
g) If *Mbin* is not a multiple of three bits long, append 0 bits to bring it up to a multiple of three.
h) Convert Mbin to a ternary polynomial of degree *N*-1 as follows. Treat Mbin as a concatenation of 3-bit quantities. Convert each three-bit quantity to two ternary coefficients as follows, and concatenate the resulting ternary quantities to obtain Mtrin.
   — {0, 0, 0} -> {0, 0}
   — {0, 0, 1} -> {0, 1}
   — {0, 1, 0} -> {0, -1}
   — {0, 1, 1} -> {1, 0}

- — {1, 0, 0} -> {1, 1}
- — {1, 0, 1} -> {1, -1}
- — {1, 1, 0} -> {-1, 0}
- — {1, 1, 1} -> {-1, 1}

i) Convert the public key *h* to a bit string *bh* using RE2BSP (7.5.1). Form the bit string *bhTrunc* by taking the first *pkLen* bits of *bh*. Convert *bhTrunc* to the octet string *hTrunc*, of length *pkLen*/8 using BS2OSP. Form *sData* as the octet string

OID || *m* || *b* || *hTrunc*

j) Use the chosen blinding polynomial generation method with the seed *sData* and the chosen parameters to produce *r* if *is_product_form* = false, and r_1, r_2 and r_3 if *is_product_form* = True. IF the blinding polynomial generation method outputs "error", output "error".

k) Calculate $R = r \times h \bmod q$. If *is_product_form* = True, use product form polynomial multiplication.

l) Calculate $R4 = R \bmod 4$.

m) Convert *R4* to the octet string *oR4* using RE2OSP, using q=4 within RE2OSP.

n) Generate a masking polynomial *mask* by calling the given MGF with inputs (*oR4*, *N*, *minCallsMask*)

o) Form m' by polynomial addition of *M* and *mask* mod p.

p) If the number of 1s, or -1s, or 0s in m' is less than d_{m0}, discard m' and return to step c).

q) Calculate the ciphertext as e = R + m' mod *q*.

r) Output e.

Graphically, the encryption operation may be represented as follows:

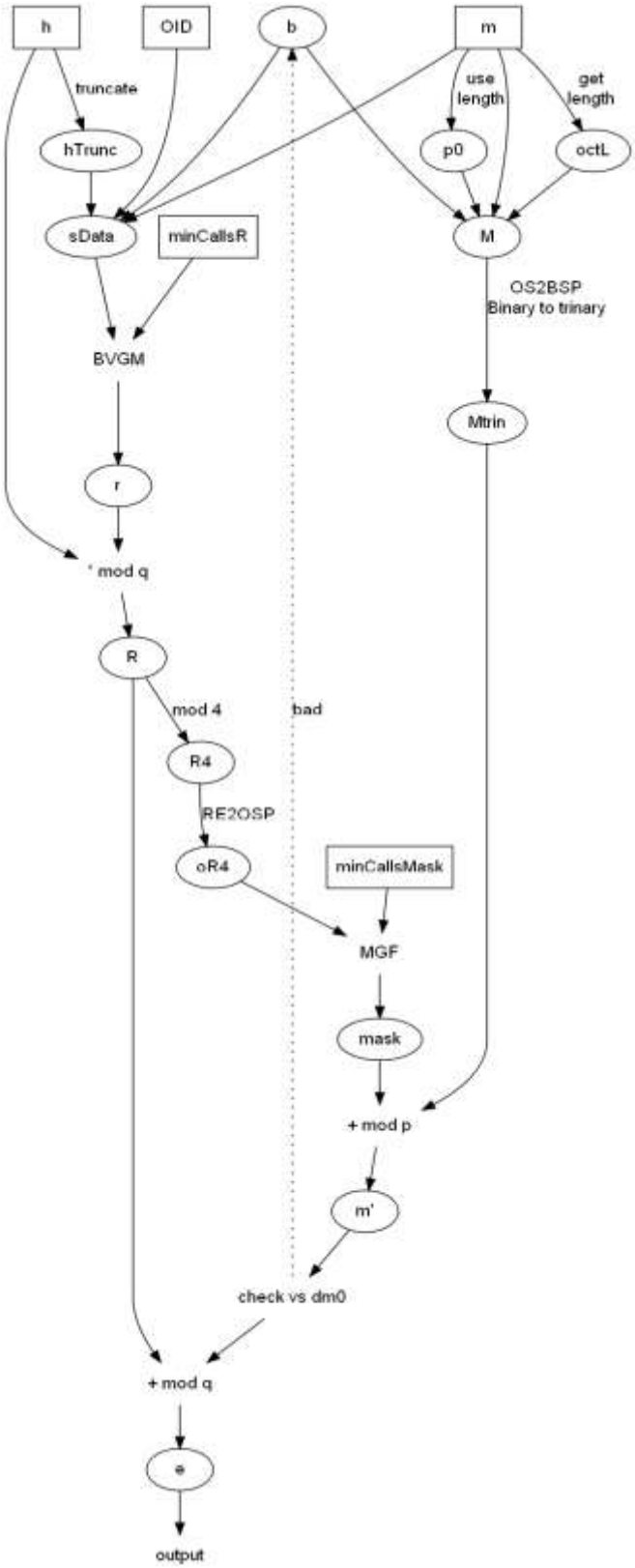

Figure 9.1: Encryption Operation

9.2.3 Decryption Operation

This section defines the decryption operation. Note that within the definition of the 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 LBP-PKE decryption primitive to use
- The length of the encoded length *lLen*.
- The number of bits of random data *db*, which shall be a multiple of 8.
- The chosen Mask Generation Function and Hash Function.
- The chosen Blinding Polynomial Generation Method and the associated parameters
- The OID, an octet string
- The number of bits of public key to hash, *pkLen*.
- The lower bound *A*
- The minimum message representative weight d_{m0}
- The maximum message length *maxMsgLenBytes*

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:

a) Calculate:
   1) *nLen* = ceil [*N*/8], the number of octets required to hold *N* bits.
   2) *bLen* = *db*/8, the length in octets of the random data
   3) *maxLen* = *nLen* - 1 - *lLen* - *bLen*, the maximum message length.

b) Decrypt the ciphertext *e* using the selected NTRU decryption primitive with inputs *e* and *f* to get the candidate decrypted polynomial *ci*.

c) If the number of 1s, or -1s, or 0s in *ci* is less than d_{m0}, set "fail" to 1.

d) Calculate the candidate value for $r \times h$, *cR* = *e* - *ci*.

e) Calculate *cR4* = *cR* mod 4.

s) Convert *cR4* to the octet string *coR4* using RE2OSP, using *q*=4 within RE2OSP.

f) Generate a masking polynomial *mask* by calling the given MGF with inputs (*coR4*, *N*, *minCallsMask*)

g) Form *cMTrin* by polynomial subtraction of *cm'* and *mask* mod p.

h) Convert *cMTrin* to a bit string as follows. Treat *cMTrin* as a concatenation of polynomials each containing 2 ternary coefficients. Convert each set of two ternary coefficients to three bits as follows, and concatenate the resulting bit quantities to obtain *cMBin*
   - {0, 0} -> {0, 0, 0}
   - {0, 1} -> {0, 0, 1}
   - {0, -1} -> {0, 1, 0}
   - {1, 0} -> {0, 1, 1}
   - {1, 1} -> {1, 0, 0}

- — {1, -1} -> {1, 0, 1}
- — {-1, 0} -> {1, 1, 0}
- — {-1, 1} -> {1, 1, 1}
- — {-1, -1} -> set "fail" to 1 and set bit string to {1, 1, 1}

i) If *cMBin* is not a multiple of 8 bits long, remove the final (length – length mod 8) bits.

j) Convert *cMBin* to an octet string *cM* using BS2OSP.

k) Parse *cM* as follows.

1) The first *bLen* octets are the octet string *cb*.

2) The next *lLen* octets represent the message length. Convert the value stored in these octets to the candidate message length *cl*. If *cl* > *maxMsgLenBytes*, set *fail* = 1 and set *cl* = *maxL*.

3) The next *cl* octets are the candidate message *cm*. the remaining octets should be 0. If they are not, set fail = 1.

l) Convert the public key *h* to a bit string *bh* using RE2BSP (7.5.1). Form the bit string *bhTrunc* by taking the first *pkLen* bits of *bh*. Convert *bhTrunc* to the octet string *hTrunc*, of length *pkLen*/8 using BS2OSP. Form *sData* as the octet string

OID || *m* || *b* || *hTrunc*

m) Use the chosen blinding polynomial generation method with the seed *sData* and the chosen parameters to produce *cr* if *is_product_form* = false, and cr_1, cr_2 and cr_3 if *is_product_form* = True. IF the blinding polynomial generation method outputs “error”, set *fail* = 1.

n) Calculate *cR*' = *h* × *cr* mod *q*. If *is_product_form* = True, use product form polynomial multiplication.

o) If *cR*' != *cR*, set *fail* = 1

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

Graphically, the encryption operation may be represented as follows:

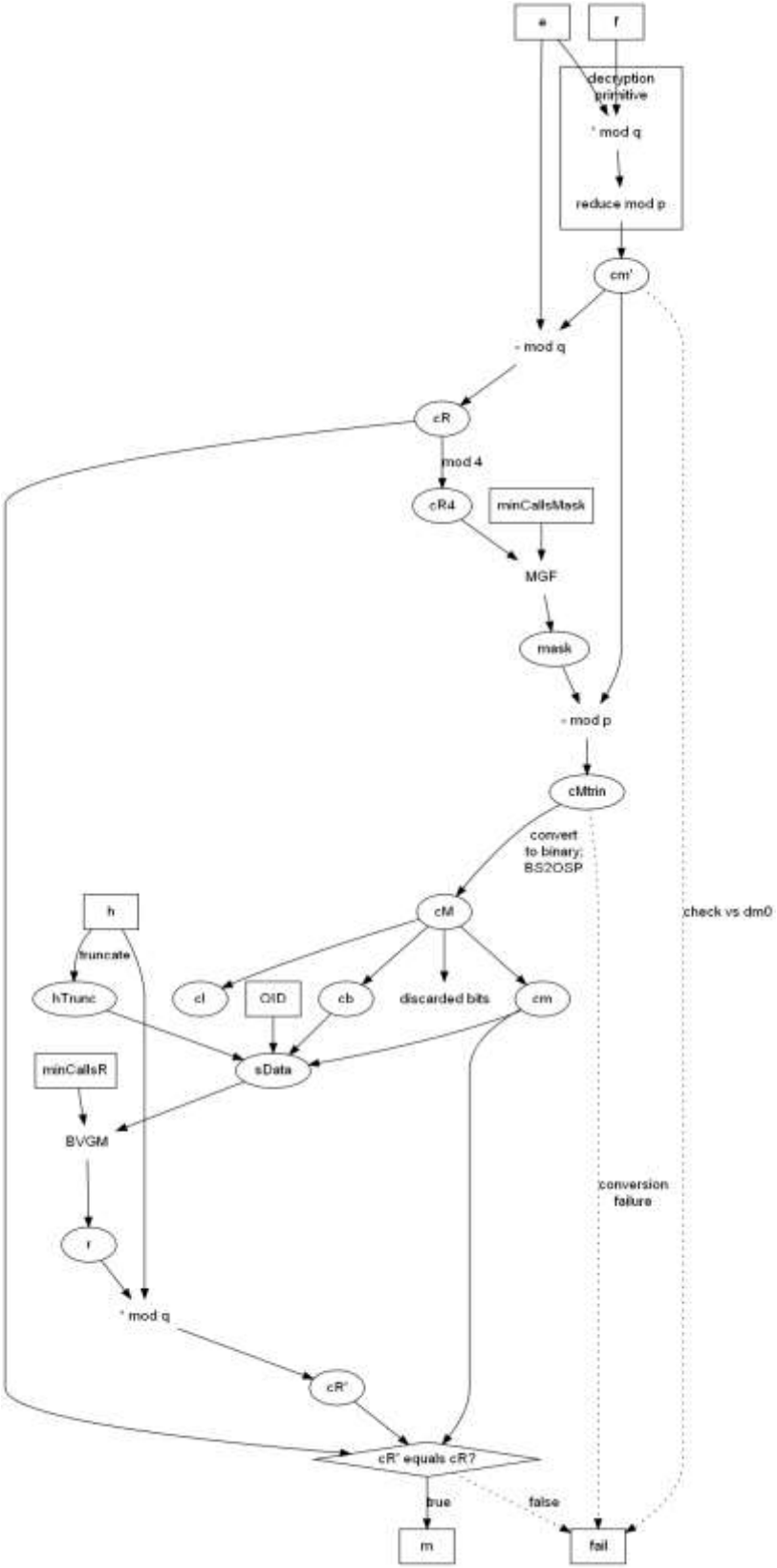

Figure 9.2: Decryption Operation

9.2.4 Key Pair Validation Methods

A key pair validation method determines whether a candidate LBP-PKE public-key/private-key pair meets the constraints for key pairs produced by a particular key generation method.

9.2.4.1 kpv3: Key Pair Validation for Ternary Keys

This key validation method corresponds to the key generation operation in 9.2.1.

Algorithm 1 – kpv3, Key Pair Validation for Ternary Keys

Components: The parameters N, q, dF, d_g,

Input: The private key component F and the public key h.

Output: "valid" or "invalid".

Operation:

a) Check that h is a polynomials of degree no greater than N-1. If not, output "invalid" and stop.

b) Check that all of the coefficients of h lie in the range [0, q-1]. If any coefficients lie outside this range, output "invalid" and stop.

c) If *is_product_form* = flase:
   1) Convert *private_key_blob* into a polynomial F.
   2) Check that F is ternary with exactly dF 1s and dF -1s. If not, output "invalid" and stop.
   3) Check that F is a polynomial of degree no greater than N-1. If not, output "invalid" and stop.
   4) Set $f = 1 + 3F \bmod q$.
   5) Set $g = f \times h \bmod q$.

d) If *is_product_form* = true:
   1) Convert *private_key_blob* into three polynomial F_1, F_2 and F_3.
   2) Check that F_1 is ternary with exactly df_1 1s and df_1 -1s. If not, output "invalid" and stop.
   3) Check that F_2 is ternary with exactly df_2 1s and df_2 -1s. If not, output "invalid" and stop.
   4) Check that F_3 is ternary with exactly df_3 1s and df_3 -1s. If not, output "invalid" and stop.
   5) Check that F_1, F_2 and F_3 are polynomials of degree no greater than N-1. If either of them has greater degree, output "invalid" and stop
   6) Set $F = F_1 \times F_2 + F_3$
   7) Compute $g = 3(F_1 \times F_2 + F_3) \times h + h \bmod q$ using product form polynomial multiplication.

e) Check that g is ternary with exactly (d_g +1) 1s and d_g -1s. If it is not, output "invalid" and stop.

f) Output "valid".

9.2.5 Public-key validation

9.2.5.1 Full public-key validation

A full public-key validation method determines whether a candidate public key satisfies the definition of a public key and meets any additional constraints imposed by a given key pair generator. Such methods provide the highest assurance to a relying party. For example, for keys generated using the key generation operation in 9.2.1, full public-key validation would prove that $h = f^{-1}g \bmod q$, where $f = 1+pF$ and F, g have dF, d_g 1s respectively. Currently there are no known methods that provide full public-key validation for the LBE-PKE schemes in this standard.

9.2.5.2 Partial public-key validation and plausibility tests

9.2.5.2.1 Overview

A partial public-key validation method determines, with some level of assurance, whether a candidate public key meets *some* of the properties of a public key. As with full public-key validation methods, partial public-key validation methods may be interactive or non-interactive. This standard supports only non-interactive methods.

Non-interactive methods for LBP-PKE public keys that do not require a witness are called *plausibility tests*. The name reflects the fact that while examining only the public key, the tests only determine whether the public key is plausible, not necessarily whether it is valid. Plausibility tests can detect unintentional errors with reasonable probability, though not with certainty. (See Note.)

This is still an active research area; further methods may be described in future versions of this Standard.

NOTE—There are other ways to detect unintentional errors; a checksum on the key can be used to detect storage and transmission errors, and the signature on a certificate will likely fail verification if the public key is modified. The checks in this section provide an additional level of assurance beyond the other methods, or an alternative when they are not available.

9.2.5.2.2 Example suite of plausibility tests

The following is an example of a plausibility test, corresponding to the key generation operation in 9.2.1.

a) Check that $h(1) = g(1)/(1 + pF(1)) \bmod q$. ($F(1) = 0$, $g(1) = 1$ so $h(1) = 1$). If it is not, output "invalid" and stop.

b) For $t = 0$ to q-1:

   1) Reduce h into the range $[t, t+q-1]$.

c) Calculate the centered norm $||h||$ for h reduced into this range.

d) Set $||h||_{min}$ equal to the minimum value of $||h||$ obtained in the previous step.

e) Set $||r|| = \sqrt{[2\ d_r]}$.

f) If $||h||_{min} > q\ (\sqrt{N}) / (3\ ||r||)$, output "plausible public key" and stop. Otherwise, output "invalid" and stop.

Steps b) to d) are motivated by the considerations that for a valid public key h, the calculation of $h \times r \bmod q$ involves a large number of reductions mod q. The test checks that $||h \times r|| > q\sqrt{(N)}/2$, in other words that the centered norm of $h \times r$ is with high likelihood greater than the centered norm of a polynomial consisting of $N/2$ coefficients with the value $q/2$ and $N/2$ coefficients with the value $–q/2$ (this calculation uses the pseudo-multiplicative property of the centered norm). For genuine h, the typical value of $||h||_{min}$ is slightly under $q\ \sqrt{(N/12)}$. For binary polynomials, the centered norm $||r||$ is $\sqrt{(2d_r)}$, which is considerably greater than $\sqrt{(3)}$ for all parameter sets in this standard. A valid h therefore passes this test with high probability.

9.3 Possible Parameter Sets

9.3.1 General

This section defines specific sets of parameters for the encryption scheme (SVES) that give a specific level of security according to the metrics in this standard.

9.3.2 ees401ep2

This parameter set is suitable for use at the 112-bit security level

Table 1 – ees401ep2
N = 401
p = 3
q = 2048
is_product_form = ture
Key generation: KGP-3 with
d_{f1} = 8
d_{f2} = 8
d_{f3} = 6
d_g = 133
lLen = 1
db = 112
maxMsgLenBytes = 60
bufferLenBits = 600
bufferLenTrits = 400
d_{m0} = 101
MGF-TP-1 with
SHA-1 (MGF)
BPGM2 with
IGF-MGF-1 with SHA-1 (IGF)
d_{r1} = 8
d_{r2} = 8
d_{r3} = 6
c = 11
minCallsR = 10
minCallsMask = 6
OID = 00 02 10
pkLen = 114

NOTE— If a message representative m' has fewer than d_{m0} 1s, -1s, or 0s, it shall be rejected. The chance of this happening with a legitimately generated m' is less than 2^{-10}.

9.3.3 ees439ep1

This parameter set is suitable for use at the 128-bit security level

Table 2 – ees439ep1
N = 439
p = 3
q = 2048
is_product_form = ture
Key generation: KGP-3 with
d_{f1} = 9
d_{f2} = 8
d_{f3} = 5
d_g = 146
lLen = 1
db = 128
maxMsgLenBytes = 65
bufferLenBits = 660
bufferLenTrits = 440
d_{m0} = 112
MGF-TP-1 with
SHA-1 (MGF)
BPGM3 with
IGF-MGF-1 with SHA-1 (IGF)
d_{r1} = 9
d_{r2} = 8
d_{r3} = 5
c = 9
minCallsR = 15
minCallsMask = 6
OID = 00 03 10
pkLen = 128

NOTE— If a message representative m' has fewer than d_{m0} 1s, -1s, or 0s, it shall be rejected. The chance of this happening with a legitimately generated m' is less than 0.000471**.**

9.3.4 ees593ep1

This parameter set is suitable for use at the 128-bit security level.

Table 3 – ees593ep1
N = 593
p = 3
q = 2048
Key generation: KGP-3 with
d_{f1} = 10
d_{f2} = 10
d_{f3} = 8
d_g = 197
lLen = 1
db = 196
maxMsgLenBytes = 86
bufferLenBits = 888
bufferLenTrits = 592
d_{m0} = 158
MGF-TP-1 with
SHA-1 (MGF)
BPGM3 with
IGF-MGF-1 with SHA-1 (IGF)
d_{r1} = 10
d_{r2} = 10
d_{r3} = 8
c = 11
minCallsR = 12
minCallsMask = 5
OID = 00 05 10
pkLen = 196

NOTE— If a message representative m' has fewer than d_{m0} 1s, -1s, or 0s, it shall be rejected. The chance of this happening with a legitimately generated m' is less than 0.000552.

9.3.5 ees743ep1

This parameter set is suitable for use at the 256-bit security level

Table 4 – ees743ep1
N = 743
p = 3
q = 2048
Key generation: KGP-3 with
d_{f1} = 11
d_{f2} = 11
d_{f3} = 15
d_g = 247
lLen = 1
db = 256
maxMsgLenBytes = 106
bufferLenBits = 1112
bufferLenTrits = 742
d_{m0} = 204
MGF-TP-1 with
SHA-1 (MGF)
BPGM3 with
IGF-MGF-1 with SHA-1 (IGF)
d_{r1} = 11
d_{r2} = 11
d_{r3} = 15
c = 13
minCallsR = 12
minCallsMask = 7
OID = 00 05 10
pkLen = 256

NOTE— If a message representative m' has fewer than d_{m0} 1s, -1s, or 0s, it shall be rejected. The chance of this happening with a legitimately generated m' is less than 0.000723.

10 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.

10.1 General Types

10.1.1 General Vector Types

This section defines the ASN.1 syntax for vector types that are used to represent polynomials for NTRUEncrypt. There are four primary types of vectors – public vectors, binary vectors, trinary vectors, and listed 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 these 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 $-q/2 <= x <= q/2$). Finally, listed vectors are an array of indices mod *N*, where each index identifies a coefficient with a known value.

NOTE: Binary vectors are not supported in this version of the standard but are included for backwards compatibility of the ASN.1 structures.

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 the integer 2) and using I2BSP (see section7.2.1) to obtain a bit string of the appropriate length. The integer is recovered by obtaining the correct bit string (e.g. for the NTRUEncrypt parameters in this standard, each coefficient of a polynomial mod q = 2048 is represented by 11 bits), and using BS2IP (see section 7.2.2). So, to encode the value 55 as an 11-bit value, the integer is encoded as the 11-bit string 000 0011 0111 (using I2BSP). To obtain the value of the coefficient represented by the bit string 000 0011 0111, the bit string is converted to the integer 55 (using BS2IP).

```
NTRUPublicVector ::= CHOICE {
        modQVector          ModQVector, -- not recommended
        packedModQVector    PackedModQVector,
        …
}

NTRUBinaryVector ::= CHOICE {
        listedBinaryVector      ListedBinaryVector,
        packedBinaryVector      PackedBinaryVector,
        modQVector              ModQVector,
        …
}
```

Binary vectors are not supported in this version of the standard.

ModQVector ::= OCTET STRING

The contents of this **OCTET STRING** are obtained by converting a ring element to an octet string using the RE2POSP 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.

PackedModQVector ::= OCTET STRING

The contents of this **OCTET STRING** are obtained by converting a ring element to an octet string using the RE2OSP 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.

This is the preferred format for public keys and ciphertexts for the parameter sets in this standard.

ListedBinaryVector ::= OCTET STRING

This type is not supported in this version of this standard.

PackedBinaryVector ::= OCTET STRING

This type is not supported in this version of this standard.

ListedTrinaryVector ::= OCTET STRING

The contents of this **OCTET STRING** are obtained by converting a trinary vector to an octet string with TRE2OSP. This is the preferred format for NTRUEncrypt private key components.

PackedTrinaryVector ::= OCTET STRING

The contents of this **OCTET STRING** are obtained by converting a trinary vector to an octet string with RE2OSP with q set equal to 3.

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

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}
```

10.2 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.

10.2.1 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 10.2.4 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               INTEGER,
        …}
```

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

- **publicKeyVector** is the polynomial h.
- **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.

10.2.2 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 10.2.4 below.

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

```
NTRUTrinaryPrivateKey ::= SEQUENCE {
        version                 INTEGER,
        publicKeyVector         NTRUPublicVector OPTIONAL,
        ntruPrivateKeyVectors CHOICE {
          productForm             NTRUProductFormTrinaryPrivateKeyVectors,
          …
        },
        …}

NTRUProductFormTrinaryPrivateKeyVectors ::= SEQUENCE {
        f1              ListedTrinaryVector,
        f2              ListedTrinaryVector,
        f3              ListedTrinaryVector,
        g               PackedTrinaryVector OPTIONAL }
```

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.
- **privateKeyVectors** contains the private key vector. The only type of private key vector supported in this standard has f = 1 + *p**(f1*f2 + f3).

10.2.3 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 10.2.4 below.

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

```
NTRUEncryptedData ::= NTRUPublicVector
```

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

10.2.4 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                  INTEGER, -- this choice is deprecated
        standardNTRUParameters  StandardNTRUParameters,
        explicitNTRUParameters  ExplicitNTRUParameters,
        externalParameters      NULL }

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

NTRUParameters OIDS ::= {
        { OID id-ees401ep2 } |
        { OID id-ees439ep1 } |
        { OID id-ees593ep1 } |
        { OID id-ees743ep1 } |
        … -- allows for future expansion
        -- other OIDs defined in previous versions of this standard are deprecated
        }

id-ees401ep2 OBJECT IDENTIFIER ::= {id-eess1-params 46}
id-ees439ep1 OBJECT IDENTIFIER ::= {id-eess1-params 47}
id-ees593ep1 OBJECT IDENTIFIER ::= {id-eess1-params 48}
id-ees743ep1 OBJECT IDENTIFIER ::= {id-eess1-params 49}
```

- **degree** is deprecated
- **standardNTRUParameters** identifies the parameters by use of an OID. In this document, four OIDs are defined: **id-ees401ep2**, **id-ees439ep1, id-ees593ep1,** and **id-ees743ep1**.
- **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).

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: 3.0 $

DEFINITIONS IMPLICIT 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}

-- parameter set identifiers

id-ees401ep2 OBJECT IDENTIFIER ::= {id-eess1-params 46}
id-ees439ep1 OBJECT IDENTIFIER ::= {id-eess1-params 47}
id-ees593ep1 OBJECT IDENTIFIER ::= {id-eess1-params 48}
id-ees743ep1 OBJECT IDENTIFIER ::= {id-eess1-params 49}

-- General types

ModQVector ::= OCTET STRING

PackedModQVector ::= OCTET STRING

ListedBinaryVector ::= OCTET STRING

PackedBinaryVector ::= OCTET STRING

ListedTrinaryVector ::= OCTET STRING

PackedTrinaryVector ::= OCTET STRING

-- NTRUEncrypt Encryption

NTRUPublicVector ::= CHOICE {
        modQVector              ModQVector, -- not recommended
        packedModQVector    PackedModQVector,
   ...
}

NTRUKeyExtension ::= CHOICE {
        keyID             INTEGER,
        ...}

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

NTRUTrinaryPrivateKey ::= SEQUENCE {
        version                 INTEGER,
        publicKeyVector         NTRUPublicVector OPTIONAL,
```

```
        ntruPrivateKeyVectors CHOICE {
          productForm          NTRUProductFormTrinaryPrivateKeyVectors,
          ...
        },
...}

NTRUProductFormTrinaryPrivateKeyVectors ::= SEQUENCE {
        f1              ListedTrinaryVector,
        f2              ListedTrinaryVector,
        f3              ListedTrinaryVector,
        g               PackedTrinaryVector OPTIONAL }

NTRUEncryptedData ::= NTRUPublicVector

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

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

NTRUParameters OIDS ::= {
{ OID id-ees401ep2 } |
{ OID id-ees439ep1 } |
{ OID id-ees593ep1 } |
{ OID id-ees743ep1 }
        … -- allows for future expansion
        -- other OIDs defined in previous versions of this standard are deprecated
        }

ExplicitNTRUParameters ::= OCTET STRING

END  -- EESS-1 --
```

Appendix B - Security Considerations

[To be added in future versions]

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
Version 3.0 available April 1, 2015