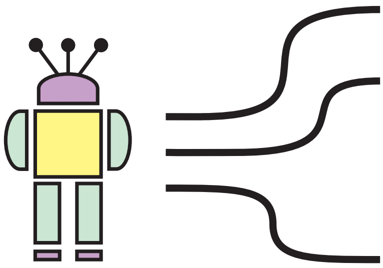
Everything is a message

MAM2 (Masked Authenticated Messaging Version 2) Protocol



Revision: 0.7
Date: 2019.02.27

Revision History

Revision	Date	Description
0.1	2018.07.30	Initial draft
0.2	2018.10.23	Add PKE algorithms (the `NTRU` layer)
		Make `oneof` alternatives absorbed (security reasons)
		Make `repeated` repetitions absorbed (security reasons)
		Disable the `optional` modifier (`optional` is covered by `oneof`)
		Disable the `required` modifier (it becomes redundant)
0.3	2018.10.31	Shorten the length of NTRU private keys (drop out g)
		Add handling of possible decoding errors during NTRU decryption
		Announce standard messages including public key certificates
0.4	2018.12.23	Refine `Sponge.Squeeze`
		Add the `Spongos` layer: cryptoprocessing of strictly formatted data
		Switch `WOTS`, `MSS`, `NTRU`, `MAM2` from `Sponge` to `Spongos`
		Change Protobuf3 cryptographic modifiers to fit `Spongos`
0.5	2019.02.06	Make `NTRU.Encr`.r dependent on public key (see 10.4)
		Insert an additional commit in `NTRU.Encr/Decr` (see 10.4, 10.5)
		Rename `Header.nonce` to `Header.msgid` (see 12.2)
		Add `Header.typeid` (see 12.2)
		Remove `KeyloadPlain` from `Header.keyload` (see 12.2)
		Make `Header.ord` negative in the last packet (see 12.2)
		Add Chapter 14 (Transport over the Tangle)
0.6	2019.02.18	Add $\langle\cdot\rangle$ notation which supports `size_t` (see 1, 11.2)
		Warn about duplication / parallel usage of names / keys (see 12.1)
		Absorb N, N' with their lengths in `MAM2.{CreateXXX|Send}`
		Choose `msgid` explicitly in `MAM2.Send` and use it for generating K
		Make MAM2 payloads / names strings of trytes, not trits (see 12)
0.7	2019.02.27	Shorten `Header.msgid` to 21 trytes (see 12)
		Change semantics of the `tag` field (see 14)

Contents

1 Notations

	miscellaneous
$\perp$	a special object or event: an empty word, an unused variable, an error;
$u \leftarrow a$	assign a value a to a variable u;
$u \xleftarrow{R} A$	choose u randomly (uniformly independently of other choices) from a set A;
$\mathtt{Alg}(a_1, a_2, \ldots)$	calling an algorithm $\mathtt{Alg}$ with inputs $a_1, a_2, \ldots$;
$\mathtt{Alg}[p](a_1, a_2, \ldots)$	calling an algorithm $\mathtt{Alg}$ with a parameter (an input with a small amount of possible values) p and inputs $a_1, a_2, \ldots$;
$\overline{a}$	the same as $-a$;
$\lfloor a \rfloor$	the maximum integer not exceeding a;
	words
T	$\{\overline{1}, 0, 1\}$, the ternary alphabet;
T^n	the set of all words of length n in T;
T^*	the set of all words of finite length in T (including the empty word $\perp$ of length 0);
T^{n*}	the set of all words from T^* whose lengths are multiplies of n;
$\lvert u \rvert$	the length of $u \in \mathsf{T}^*$;
α^n	for $\alpha \in \mathsf{T}$, the word of n instances of α ($\alpha^0 = \perp$);
$u[i]$	the i-th trit of $u \in \mathsf{T}^n$: $u = u[0]u[1]\ldots u[n-1]$;
$u[\ldots l)$	for $u \in \mathsf{T}^n$ and $0 < l \leqslant n$, the subword $u[0]u[1]\ldots u[l-1]$;
$u[l\ldots)$	for $u \in \mathsf{T}^n$ and $0 \leqslant l < n$, the subword $u[l]u[l+1]\ldots u[n-1]$;
$u[l_1\ldots l_2)$	for $u \in \mathsf{T}^n$ and $0 \leqslant l_1 < l_2 \leqslant n$, the subword $u[l_1]u[l_1+1]\ldots u[l_2-1]$;
	integers
$U \bmod m$	for an integer U and a positive integer m, the unique $r \in \{0, 1, \ldots, m-1\}$ such that m divides $U - r$;
$U \operatorname{mods} m$	for an integer U and a positive odd integer m, the unique $r \in \left\{-\frac{m-1}{2}, -\frac{m-3}{2}, \ldots, \frac{m-1}{2}\right\}$ such that m divides $U - r$;
$[u]$	for $u \in \mathsf{T}^n$ the integer $U = u[0] + 3u[1] + \ldots + 3^{n-1}u[n-1]$;
$\langle U \rangle_n$	for an integer U and a positive integer n, the word $u \in \mathsf{T}^n$ such that $[u] = U \operatorname{mods} 3^n$;
$\langle U \rangle$	for an integer U, the word $u = \langle n \rangle_3 \parallel \langle U \rangle_{3n}$, where n is the minimum positive integer such that u unambiguously encodes U;
	operations
$u \parallel v$	the concatenation of $u, v \in \mathsf{T}^*$: a word $w \in \mathsf{T}^{\lvert u \rvert + \lvert v \rvert}$ such that $w[\ldots \lvert u \rvert) = u$ and $w[\lvert u \rvert \ldots) = v$;
$u \oplus v$	for $u, v \in \mathsf{T}^n$, the word $w \in \mathsf{T}^n$ in which $w[i] = (u[i] + v[i]) \operatorname{mods} 3$;
$u \ominus v$	for $u, v \in \mathsf{T}^n$, the word $w \in \mathsf{T}^n$ such that $u = v \oplus w$;
	crypto
F	a bijective function $\mathsf{T}^{729} \to \mathsf{T}^{729}$ (sponge function) defined outside this specification.

2 Examples

$$
\begin{aligned}
|\overline{1}0\overline{1}11\overline{1}| &= 6 \\
\overline{1}0\overline{1}11\overline{1}[\ldots 4) &= \overline{1}0\overline{1}1 \\
\overline{1}0\overline{1}11\overline{1}[2\ldots) &= \overline{1}11\overline{1} \\
\overline{1}0\overline{1}11\overline{1}[2\ldots 4) &= \overline{1}1 \\
[\overline{1}0\overline{1}11\overline{1}] &= -1-9+27+81-243 = -145 = \overline{145} \\
\langle\overline{145}\rangle_6 &= \overline{1}0\overline{1}11\overline{1} \\
[01\overline{11}11] &= 3-9-27+81+243 = 291 \\
\langle 291\rangle_5 &= 01\overline{11}1 \\
\langle 291\rangle_6 &= 01\overline{11}11 \\
\langle 291\rangle_7 &= 01\overline{11}110 \\
\langle 291\rangle &= \overline{1}10 \parallel 01\overline{11}11 = \overline{1}1001\overline{11}11 \\
\overline{1}0\overline{1}11\overline{1} \parallel 01\overline{11}11 &= \overline{1}0\overline{1}11\overline{1}01\overline{11}11 \\
\overline{1}0\overline{1}11\overline{1} \oplus 01\overline{11}11 &= \overline{1}110\overline{1}0 \\
\overline{1}0\overline{1}11\overline{1} \ominus 01\overline{11}11 &= \overline{11}0\overline{1}01
\end{aligned}
$$

3 Glossary

3.1 trit: an element of T;
3.2 trint: a word of three trytes interpreted as an integer from the set $\{\overline{9841}, \ldots, 9841\}$;
3.3 tryte: a word of three trits interpreted as an integer from the set $\{\overline{13}, \ldots, 13\}$. Trytes are encoded by symbols of the alphabet $\{\mathtt{9}, \mathtt{A}, \ldots, \mathtt{Z}\}$ in accordance with Table 1;

Table 1: Trytes

tryte	integer	code	tryte	integer	code	tryte	integer	code
000	0	`9`	001	9	`I`	$00\overline{1}$	$\overline{9}$	`R`
100	1	`A`	101	10	`J`	$10\overline{1}$	$\overline{8}$	`S`
$\overline{1}10$	2	`B`	$\overline{1}11$	11	`K`	$\overline{1}1\overline{1}$	$\overline{7}$	`T`
010	3	`C`	011	12	`L`	$01\overline{1}$	$\overline{6}$	`U`
110	4	`D`	111	13	`M`	$11\overline{1}$	$\overline{5}$	`V`
$\overline{11}1$	5	`E`	$\overline{111}$	$\overline{13}$	`N`	$\overline{11}0$	$\overline{4}$	`W`
$0\overline{1}1$	6	`F`	$0\overline{11}$	$\overline{12}$	`O`	$0\overline{1}0$	$\overline{3}$	`X`
$1\overline{1}1$	7	`G`	$1\overline{11}$	$\overline{11}$	`P`	$1\overline{1}0$	$\overline{2}$	`Y`
$\overline{1}01$	8	`H`	$\overline{1}0\overline{1}$	$\overline{10}$	`Q`	$\overline{1}00$	$\overline{1}$	`Z`

3.4 channel: a source of messages. Belongs to an entity. Identified by a public key, called `chid`, corresponding to which private keys are used to sign either endpoints (main functionality) or messages;

3.5 endpoint: a transmitter of messages. Belongs to a channel. Identified by a public key, called `epid`, corresponding to which private keys are used to sign messages;
3.6 central endpoint: an endpoint which `epid` is equal to `chid` of the corresponding channel;
3.7 layer: a set of interconnected algorithms. The algorithms share a common state which is implicitly included in their inputs and outputs (that is, each algorithm can use and modify the common state);
3.8 nonce: an input to a cryptographic algorithm which is unique for sure or with an overwhelming probability.

4 Preliminaries

This document is a specification of MAM2 (Masked Authenticated Messaging version 2) cryptographic protocol designated for the IOTA framework.

Using MAM2 entities of IOTA can (see Figure 1):

– create channels for broadcasting messages;

– create channel endpoints for protecting messages during broadcasting;

– protect messages in different ways, for example, turn on / off encryption / authentication;

– split messages into parts (packets), protect and transmit each part almost independently;

– set message recipients and provide them with key material in different ways.

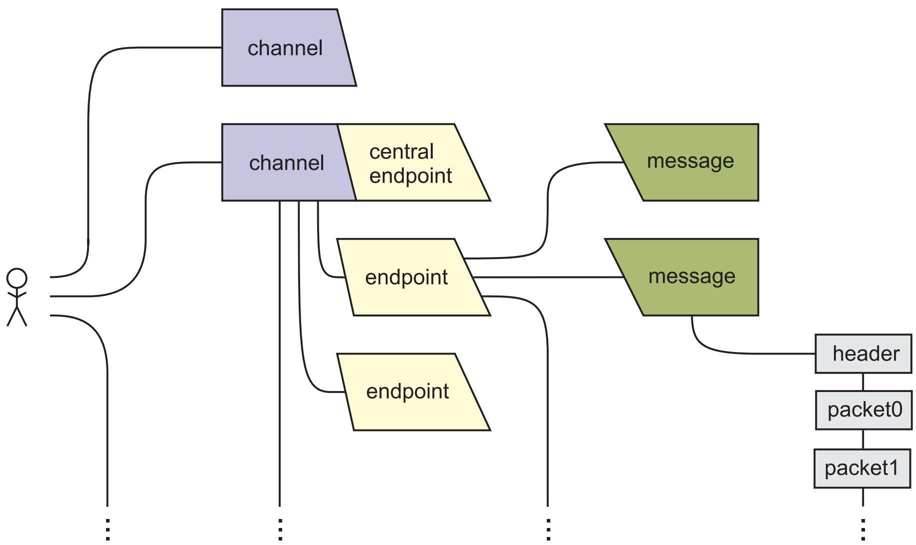


Figure 1: The concept of MAM2

Cryptographic algorithms of MAM2 are based on a sponge function F, which is determined outside of this specification. Algortithms are grouped into layers. They are

– `Sponge` (basic cryptographic processing of loosely formatted data);

– `Spongos` (basic cryptographic processing of strictly formatted data);

– `WOTS` (one-time signatures based on a hash algorithm from `Sponge`);

– `MSS` (multi-time signatures over `WOTS`);

– `NTRU` (public key encryption).

Additional layers are

– `Protobuf3` (encoding, decoding and high-level cryptographic processing of messages);

– `MAM2` (the overall protocol).

The `Sponge` and `Spongos` (from $\sigma\pi o\gamma\gamma o\varsigma$, "sponge" in Greek) are essentially two "languages" for cryptographic data processing using F. The first layer is intended for working with loosely defined formats: when we start to process a data field we know exactly how to process its successive trits, but we may not know which trit will be the last. The second layer is intended for working with strictly defined formats: we know exactly both the way of processing and the field length.

5 The `Sponge` layer

5.1 Overview

The `Sponge` layer supports operations based on a sponge function $F\colon \mathsf{T}^{729} \to \mathsf{T}^{729}$. Basic operations of the layer process a single block of input data or (and) create a single block of output data. Compound operations process a series of input blocks or (and) create a series of output blocks.

The layer state is a word $S \in \mathsf{T}^{729}$. The state is changed during the operations.

The state is divided into three parts:

1. The rate part $S[\ldots 486)$. It is updated by input blocks or (and) determines output blocks.

2. The control part $S[486\ldots 492)$. It consists of two control trytes: $S[486\ldots 489)$ and $S[489\ldots 492)$. The first (second) control tryte describes the previous (current) basic sponge operation.

3. The capacity part $S[492\ldots)$. It is never outputted and is changed only by applying F to the previous state.

Trits of the control tryte $c[0]c[1]c[2]$ have the following meaning (see Table 2):

– $c[0]$ describes a block of input data processed during the target operation;

– $c[1]$ describes a type of block of input or output data in the scope of the outer compound operation;

– $c[2]$ describes a type of the operation.

The mnemonic names from Table 2 can be used as $c[2]$ values in appropriate cases. For example, `KEY` can be used instead of 0 if $c[0] = 0$ or 1.

The layer contains the following algorithms:

Table 2: Control trits

trit	value	meaning	mnemonic name
$c[0]$	0	An incomplete input block	
	1	A complete input block	
	$\bar{1}$	An output block	
$c[1]$	0	A block of the initial state	
	1	An intermediate (not last) block	
	$\bar{1}$	A last block	
$c[2]$	0	Processing public data (if $c[0] \neq \bar{1}$)	`DATA`
	0	Generating a hash value (if $c[0] = \bar{1}$)	`HASH`
	1	Processing a secret key (if $c[0] \neq \bar{1}$)	`KEY`
	1	Generating pseudorandom numbers (if $c[0] = \bar{1}$)	`PRN`
	$\bar{1}$	Processing plaintext or ciphertext(if $c[0] \neq \bar{1}$)	`TEXT`
	$\bar{1}$	Generating a MAC (if $c[0] = \bar{1}$)	`MAC`

– `Init` (initialize a state, 5.2);

– `Absorb` (process input data, 5.3);

– `Squeeze` (generate output data, 5.4);

– `Hash` (hashing, 5.5);

– `Encr` (encrypt plaintext, 5.6);

– `Decr` (decrypt ciphertext, 5.7).

5.2 `Init`

Input: $\perp$.

Output: $\perp$.

Steps:

1. $S \leftarrow 0^{729}$.

5.3 `Absorb`

Parameters: $c \in \mathbf{T}$ (`DATA` or `KEY`).

Input: $X \in \mathbf{T}^*$ (input data).

The input X is divided into the blocks $X_1, X_2, \ldots, X_n \in \mathbf{T}^*$ such that $|X_1| = \ldots = |X_{n-1}| = 486$, $0 \leqslant |X_n| \leqslant 486$. The last block X_n can be empty only if X is empty (in this case $n = 1$).

Output: $\perp$.

Steps:

1. For $i = 1, 2, \ldots, n$:

   1) if $|X_i| = 486$, then $t_0 \leftarrow 1$, else $t_0 \leftarrow 0$;
   2) if $i = n$, then $t_1 \leftarrow \bar{1}$, else $t_1 \leftarrow 1$;
   3) if $S[487] \neq 0$, then
      (a) $S[489 \ldots 492) \leftarrow t_0 t_1 c$;
      (b) $S \leftarrow F(S)$;
   4) $S[\ldots 487) \leftarrow X_i \parallel 1 \parallel 0^{486-|X_i|}$;
   5) $S[487 \ldots 489) \leftarrow t_1 c$.

5.4 Squeeze

Parameters: $c \in \mathbf{T}$ (HASH, PRN or MAC).

Input: l (a number of output trits).

Output: $Y \in \mathbf{T}^l$.

The output is constructed from the blocks $Y_1, Y_2, \ldots, Y_n \in \mathbf{T}^*$ such that $|Y_1| = \ldots = |Y_{n-1}| = 486$, $0 \leqslant |Y_n| \leqslant 486$. The last block Y_n can be empty only if $l = 0$ (in this case $n = 1$).

Steps:

1. For $i = 1, 2, \ldots, n$:

   1) if $i = n$, then $t_1 \leftarrow \bar{1}$, else $t_1 \leftarrow 1$;
   2) $S[489 \ldots 492) \leftarrow \bar{1} t_1 c$;
   3) $S \leftarrow F(S)$;
   4) $Y_i \leftarrow S[\ldots |Y_i|)$;
   5) if $|Y_i| = 486$, then $S[\ldots 486) \leftarrow 0^{486}$, else $S[\ldots 486) \leftarrow 0^{|Y_i|} \parallel 1 \parallel 0^{485-|Y_i|}$;
   6) $S[486 \ldots 489) \leftarrow \bar{1} t_1 c$.

2. Return $Y_1 \parallel Y_2 \parallel \ldots \parallel Y_n$.

5.5 Hash

Parameters: l (a length of a hash value).

Input: $X \in \mathbf{T}^*$ (data to hash).

Output: $Y \in \mathbf{T}^l$ (a hash value).

Steps:

1. `Init`$(\perp)$.

2. `Absorb`[DATA](X).

3. $Y \leftarrow$ `Squeeze`[HASH](l).

4. Return Y.

5.6 `Encr`

Input: $X \in \mathbf{T}^*$ (plaintext).

The input X is divided into the blocks $X_1, X_2, \ldots, X_n \in \mathbf{T}^*$ such that $|X_1| = \ldots = |X_{n-1}| = 486$, $0 \leqslant |X_n| \leqslant 486$. The last block $|X_n|$ can be empty only if X is empty (in this case $n = 1$).

Output: $Y \in \mathbf{T}^{|X|}$ (ciphertext).

The output is constructed from the blocks $Y_1, Y_2, \ldots, Y_n \in \mathbf{T}^*$ such that $|Y_i| = |X_i|$.

Steps:

1. For $i = 1, 2, \ldots, n$:
   1) if $|X_i| = 486$, then $t_0 \leftarrow 1$, else $t_0 \leftarrow 0$;
   2) if $i = n$, then $t_1 \leftarrow \overline{1}$, else $t_1 \leftarrow 1$;
   3) $S[489 \ldots 492) \leftarrow t_0 t_1 \overline{1}$;
   4) $S \leftarrow F(S)$;
   5) $Y_i \leftarrow X_i \oplus S[\ldots |X_i|)$;
   6) $S[\ldots 487) \leftarrow X_i \parallel 1 \parallel 0^{486-|X_i|}$;
   7) $S[487 \ldots 489) \leftarrow t_1 \overline{1}$.
2. Return $Y_1 \parallel Y_2 \parallel \ldots \parallel Y_n$.

5.7 `Decr`

Input: $Y \in \mathbf{T}^*$ (ciphertext).

The input Y is divided into the blocks $Y_1, Y_2, \ldots, Y_n \in \mathbf{T}^*$ such that $|Y_1| = \ldots = |Y_{n-1}| = 486$, $0 \leqslant |Y_n| \leqslant 486$. The last block $|Y_n|$ can be empty only if Y is empty (in this case $n = 1$).

Output: $X \in \mathbf{T}^{|Y|}$ (plaintext).

The output is constructed from the blocks $X_1, X_2, \ldots, X_n \in \mathbf{T}^*$ such that $|X_i| = |Y_i|$.

Steps:

1. For $i = 1, 2, \ldots, n$:
   1) if $|Y_i| = 486$, then $t_0 \leftarrow 1$, else $t_0 \leftarrow 0$;
   2) if $i = n$, then $t_1 \leftarrow \overline{1}$, else $t_1 \leftarrow 1$;
   3) $S[489 \ldots 492) \leftarrow t_0 t_1 \overline{1}$;
   4) $S \leftarrow F(S)$;
   5) $X_i \leftarrow Y_i \ominus S[\ldots |Y_i|)$;
   6) $S[\ldots 487) \leftarrow X_i \parallel 1 \parallel 0^{486-|X_i|}$;
   7) $S[487 \ldots 489) \leftarrow t_1 \overline{1}$.
2. Return $X_1 \parallel X_2 \parallel \ldots \parallel X_n$.

6 The `Spongos` layer

6.1 Overview

The `Spongos` layer supports operations based on a sponge function $F\colon \mathsf{T}^{729} \to \mathsf{T}^{729}$. Basic operations of the layer process a single block of input data or (and) create a single block of output data. Compound operations process a series of input blocks or (and) create a series of output blocks.

The layer state is a word $S \in \mathsf{T}^{729}$ and an index $pos \in \{0, 1, \ldots, 486\}$. The state is changed during the operations.

The word S is divided into two parts:

1. The rate part $S[\ldots 486)$. It is updated by input blocks or (and) determines output blocks.
2. The capacity part $S[486 \ldots)$. It is never outputted and is changed only by applying F to the previous state.

The layer contains the following algorithms:

– `Init` (initialize a state, 6.2);

– `Fork` (create an equivalent instance, 6.3);

– `Commit` (commit changes in the rate part, 6.4);

– `Absorb` (process input data, 6.5);

– `Squeeze` (generate output data, 6.6);

– `Hash` (hashing, 6.7);

– `Encr` (encrypt plaintext, 6.8);

– `Decr` (decrypt ciphertext, 6.9).

6.2 `Init`

Input: $\perp$.

Output: $\perp$.

Steps:

1. $S \leftarrow 0^{729}$.
2. $pos \leftarrow 0$.

6.3 `Fork`

Input: $\perp$.

Output: `spongos`$'$ (another instance of `Spongos`).

Steps:

1. Create an instance `spongos`$'$ with the state (S, pos).
2. Return `spongos`$'$.

6.4 `Commit`

Input: $\perp$.

Output: $\perp$.

Steps:

1. If $pos \neq 0$:
   1) $S \leftarrow F(S)$;
   2) $pos \leftarrow 0$.

6.5 `Absorb`

Input: $X \in \mathsf{T}^*$ (input data).

Output: $\perp$.

Steps:

1. For $i = 0, 1, \ldots, |X| - 1$:
   1) $S[pos] \leftarrow X[i]$;
   2) $pos \leftarrow pos + 1$;
   3) if $pos = 486$, then `Commit`$(\perp)$.

6.6 `Squeeze`

Input: l (a number of output trits).

Output: $Y \in \mathsf{T}^l$.

Steps:

1. $Y \leftarrow 0^l$.
2. For $i = 0, 1, \ldots, l - 1$:
   1) $Y[i] \leftarrow S[pos]$;

2) $S[pos] \leftarrow 0$;
3) $pos \leftarrow pos + 1$;
4) if $pos = 486$, then `Commit`$(\perp)$.

3. Return Y.

6.7 `Hash`

Parameters: l (a length of a hash value).

Input: $X \in \mathsf{T}^*$ (data to hash).

Output: $Y \in \mathsf{T}^l$ (a hash value).

Steps:

1. `Init`$(\perp)$.
2. `Absorb`(X).
3. $Y \leftarrow$ `Squeeze`(l).
4. Return Y.

6.8 `Encr`

Input: $X \in \mathsf{T}^*$ (plaintext).

Output: $Y \in \mathsf{T}^{|X|}$ (ciphertext).

Steps:

1. $Y \leftarrow 0^{|X|}$.
2. For $i = 0, 1, \ldots, |X| - 1$:
    1) $Y[i] \leftarrow X[i] \oplus S[pos]$;
    2) $S[pos] \leftarrow X[i]$;
    3) $pos \leftarrow pos + 1$;
    4) if $pos = 486$, then `Commit`$(\perp)$.
3. Return Y.

6.9 `Decr`

Input: $Y \in \mathsf{T}^*$ (ciphertext).

Output: $X \in \mathsf{T}^{|Y|}$ (plaintext).

Steps:

1. $X \leftarrow 0^{|Y|}$.

2. For $i = 0, 1, \ldots, |X| - 1$:

   1) $X[i] \leftarrow Y[i] \ominus S[pos]$;

   2) $S[pos] \leftarrow X[i]$;

   3) $pos \leftarrow pos + 1$;

   4) if $pos = 486$, then `Commit`$(\perp)$.

3. Return X.

7 The `PRNG` layer

7.1 Overview

The `PRNG` layer supports the generation of cryptographically strong pseudorandom numbers or, more precisely, strings of trytes. The layer makes calls to the `Sponge` layer.

The layer state is a secret key $K \in \mathbf{T}^{243}$. The key K is set when an instance of the layer is initialized. Each instance must use its own key.

The key K must be generated outside MAM2 using a strong random number generator or another pseudorandom generator with a secret key which length is not less than length of K.

There exists a global initialized instance of the `PRNG` layer. This instance, called `prng`, can be used in other layers.

The resulting pseudorandom numbers can be used in different contexts. A destination context is encoded by one tryte called a destination tryte. Allowed destination trytes are listed in Table 3.

Table 3: Destination trytes

tryte	destination	mnemonic name
`9`	secret keys	`SECKEY`
`A`	WOTS private keys	`WOTSKEY`
`B`	NTRU private keys	`NTRUKEY`

The layer contains the following algorithms:

– `Init` (initialize a state, 7.2);

– `Gen` (generate pseudorandom numbers, 7.3).

During generation of pseudorandom numbers, the state key K is used along with a destination tryte d. Additionally, a nonce $N \in \mathbf{T}^*$ is used. Different nonces N must be used with any given pair (K, d).

7.2 `Init`

Input: $X \in \mathsf{T}^{243}$ (an external key).

Output: $\perp$.

Steps:

1. $K \leftarrow X$.

7.3 `Gen`

Parameters: $d \in \mathsf{T}^3$ (a destination tryte).

Input: $N \in \mathsf{T}^*$ (a nonce), n (a number of output trits).

Output: $Y \in \mathsf{T}^n$ (pseudorandom numbers).

Steps:

1. `Sponge.Init`$(\perp)$.
2. `Sponge.Absorb`[`KEY`]$(K \parallel d \parallel N)$.
3. $Y \leftarrow$ `Sponge.Squeeze`[`PRN`](n).
4. Return Y.

8 The `WOTS` layer

8.1 Overview

The `WOTS` layer supports Winternitz One-Time Signatures.

The layer makes calls to the `Spongos` layer and to the global instance `prng` of the `PRNG` layer (see 7.1). The `prng` must be pre-initialized.

The layer state is a private key $sk \in \mathsf{T}^{13122}$. The key must be kept in secret. The corresponding public key pk, on the contrary, is publicly announced.

The key sk is deterministically generated using `prng`. Since sk is rather lengthy, it may not be stored but regenerated.

The layer contains the following algorithms:

– `Gen` (generate keys, 8.2);

– `Sign` (generate a signature, 8.3);

– `Recover` (recover a presumed public key from a signature, 8.4);

– `Verify` (verify a signature, 8.5).

8.2 `Gen`

Input: $N \in \mathsf{T}^*$ (a nonce).

Output: $pk \in \mathsf{T}^{243}$ (a public key).

Steps:

1. $sk \leftarrow \texttt{prng.Gen}[\texttt{WOTSKEY}](N, 13122)$.
2. $pk \leftarrow \perp$.
3. For $i = 1, 2, \ldots, 81$:
   1) $t \leftarrow sk[162(i-1) \ldots 162i)$;
   2) for $i = 1, 2, \ldots, 26$:
      (a) $t \leftarrow \texttt{Spongos.Hash}[162](t)$;
   3) $pk \leftarrow pk \parallel t$.
4. $pk \leftarrow \texttt{Spongos.Hash}[243](pk)$.
5. Return pk.

8.3 `Sign`

Input: $H \in \mathsf{T}^{234}$ (a hash value or MAC to be signed).

Output: $S \in \mathsf{T}^{13122}$ (a signature).

Steps:

1. $S \leftarrow \perp$.
2. $t \leftarrow 0$.
3. For $i = 1, 2, \ldots, 78$:
   1) $t \leftarrow t + [X[3(i-1) \ldots 3i)]$.
4. $h \leftarrow H \parallel \langle -t \rangle_9$.
5. For $i = 1, 2, \ldots, 81$:
   1) $s \leftarrow sk[162(i-1) \ldots 162i)$;
   2) for $j = 0, 1, \ldots, 13 + [h[3(i-1) \ldots 3i)]$:
      (a) $s \leftarrow \texttt{Spongos.Hash}[162](s)$;
   3) $S \leftarrow S \parallel s$.
6. Return S.

8.4 `Recover`

Input: $H \in \mathsf{T}^{234}$ (a signed hash value or MAC), $S \in \mathsf{T}^{13122}$ (a signature).

Output: $pk \in \mathsf{T}^{243}$ (a presumed public key).

Steps:

1. $t \leftarrow 0$.
2. For $i = 1, 2, \ldots, 78$:
   1) $t \leftarrow t + [H[3(i-1) \ldots 3i)]$.
3. $h \leftarrow H \parallel \langle -t \rangle_9$.
4. $pk \leftarrow \perp$.
5. For $i = 1, 2, \ldots, 81$:
   1) $s \leftarrow S[162(i-1) \ldots 162i)$;
   2) for $j = 0, 1, \ldots, 13 - [h[3(i-1) \ldots 3i)]$:
      (a) $s \leftarrow \texttt{Spongos.Hash}[162](s)$;
   3) $pk \leftarrow pk \parallel s$.
6. $pk \leftarrow \texttt{Spongos.Hash}[243](pk)$.
7. Return pk.

8.5 `Verify`

Input: $H \in \mathsf{T}^{234}$ (a signed hash value or MAC), $S \in \mathsf{T}^{*}$ (a signature), $pk \in \mathsf{T}^{243}$ (a public key).

Output: 1 (the signature is valid) or 0 (invalid).

Steps:

1. If $|S| \neq 13122$, then return 0.
2. Return 1, if $pk = \texttt{Recover}(H, S)$, and 0 otherwise.

9 The `MSS` layer

9.1 Overview

The `MSS` layer supports Merkle-tree Signature Scheme. Using this scheme, a signer can generate 2^d signatures of different messages.

Here d, called a height, is a parameter of the layer. It is asserted that $d \leqslant 20$ and, therefore, the numbers d and $2^d - 1$ can be represented by 4 and 14 trits respectively.

The layer makes calls to the `Spongos` and `WOTS` layers.

The layer state includes:

- a height d;
- 2^d instances of `WOTS`, denoted as $\texttt{wots}[0], \texttt{wots}[1], \ldots, \texttt{wots}[2^d - 1]$;
- a number skn of the first instance that has not yet been used for signing (or 2^d if all leaves are spent);
- a Merkle tree represented as a triangular array $mt[k, i]$, $0 \leqslant k \leqslant d$, $0 \leqslant i < 2^k$. Elements of the array (vertices of the tree) are from $\mathbf{T}^{243}$.

The `WOTS` instances contain private keys and, therefore, must be kept in secret. The private keys are deterministically generated using the global `prng` object (see 8.1). Since private keys are rather lengthy, an instance $\texttt{wots}[i]$ may not be stored but regenerated if necessary.

In the Merkle tree, the vertices $mt[k, i]$, $0 \leqslant i < 2^k$, form a level k. If $k < d$, then a vertex $mt[k, i]$ is connected with vertices $mt[k + 1, 2i]$, $mt[k + 1, 2i + 1]$ of the next level. Vertices $mt[d, i]$ are called *leaves*, the vertex $mt[0, 0]$ is called a *root*. Leaves are public keys of underlying `WOTS` instances, the root stands as the public key of the whole `MSS` layer.

There exists a single path from a leaf $mt[d, i]$ to the root $mt[0, 0]$. It has the form:

$$mt[d, i_d], mt[d - 1, i_{d-1}], \ldots, mt[1, i_1], mt[0, 0],$$

where $i_d = i$ and $i_k = \lfloor i_{k+1}/2 \rfloor$, $k = d - 1, \ldots, 2, 1$. The corresponding sequence

$$mt[d, j_d], mt[d - 1, j_{d-1}], \ldots, mt[1, j_1],$$

where

$$j_k = \begin{cases} i_k + 1, & i_k \text{ is even}, \\ i_k - 1, & i_k \text{ is odd}, \end{cases}$$

is called the *authentication path* for $mt[d, i]$.

A Merkle tree is stored in the `MSS` state to build authentication paths that are used during a signing. Since the tree can be very lengthy ($243 \cdot (2^{d+1} - 1)$ trits to store mt), several techniques to reduce the amount of memory by complicating algorithms to build authentication paths were developed. Although we do not use these techniques in this specification, they are welcomed in its implementations.

The layer contains the following algorithms:

- `Gen` (generate keys, 9.2);
- `Skn` (return d and skn, 9.3);
- `APath` (build an authentication path, 9.4);
- `Sign` (generate a signature, 9.5);
- `Verify` (verify a signature, 9.6).

9.2 `Gen`

Input: d (a height), $N \in \mathsf{T}^*$ (a nonce).

Output: $pk \in \mathsf{T}^{243}$ (a public key, a root of an internal Merkle tree).

Steps:

1. For $i = 0, 1, \ldots, 2^d - 1$:
   1) $mt[d, i] \leftarrow \texttt{wots}[i].\texttt{Gen}(N \parallel \langle i \rangle_6)$;
2. For $k = d - 1, \ldots, 1, 0$:
   1) for $i = 0, 1, \ldots, 2^k - 1$:
      (a) $mt[k, i] \leftarrow \texttt{Spongos.Hash}[243](mt[k + 1, 2i] \parallel mt[k + 1, 2i + 1])$.
3. $skn \leftarrow 0$.
4. Return $mt[0, 0]$.

9.3 `Skn`

Input: $\perp$.

Output: $Skn \in \mathsf{T}^{18}$ (encoded d and skn).

Steps:

1. Return $\langle d \rangle_4 \parallel \langle skn \rangle_{14}$.

9.4 `APath`

Input: $i \in \{0, 1, \ldots, 2^d - 1\}$ (a number of a `WOTS` instance).

Output: $p \in \mathsf{T}^{243d}$ (an authentication path).

Steps:

1. $p \leftarrow \perp$.
2. For $k = d, \ldots, 2, 1$:
   1) if i is even, then $p \leftarrow p \parallel mt[k, i + 1]$, else $p \leftarrow p \parallel mt[k, i - 1]$;
   2) $i \leftarrow \lfloor i/2 \rfloor$.
3. Return p.

9.5 `Sign`

Input: $H \in \mathsf{T}^{234}$ (a hash value or MAC to be signed).

Output: $S \in \mathsf{T}^{18+13122+243d}$ (a signature) or $\perp$ (private keys are exhausted).

Steps:

1. If $skn = 2^d$, then return $\perp$.
2. $S \leftarrow \texttt{Skn}(\perp)$.
3. $S \leftarrow S \parallel \texttt{wots}[skn].\texttt{Sign}(H)$.
4. $S \leftarrow S \parallel \texttt{APath}(skn)$.
5. $skn \leftarrow skn + 1$.
6. Return S.

9.6 `Verify`

Input: $H \in \mathsf{T}^{234}$ (a signed hash value or MAC), $S \in \mathsf{T}^*$ (a signature), $pk \in \mathsf{T}^{243}$ (a public key).

Output: 1 (the signature is valid) or 0 (invalid).

Steps:

1. If $|S| < 18 + 13122$, then return 0.
2. $d \leftarrow [S[\ldots 4)]$.
3. $skn \leftarrow [S[4 \ldots 18)]$.
4. If $d < 0$ or $skn < 0$ or $skn \geqslant 2^d$ or $|S| \neq 18 + 13122 + 243d$, then return 0.
5. $t \leftarrow \texttt{WOTS.Recover}(H, S[18 \ldots 18 + 13122))$.
6. $p \leftarrow S[18 + 13122 \ldots)$.
7. For $k = 1, 2, \ldots, d$:
   1) if skn is even, then $t \leftarrow t \parallel p[\ldots 243)$, else $t \leftarrow p[\ldots 243) \parallel t$;
   2) $t \leftarrow \texttt{Spongos.Hash}[243](t)$;
   3) $p \leftarrow p[243 \ldots)$;
   4) $skn \leftarrow \lfloor skn/2 \rfloor$.
8. Return 1, if $t = pk$, and 0 otherwise.

10 The NTRU layer

10.1 Overview

The NTRU layer supports an NTRU-style public key encryption scheme. Using NTRU a sender can encrypt session keys with a public key of a recipient.

The layer makes calls to the Spongos layer and to the global instance prng of the PRNG layer (see 7.1). The prng must be pre-initialized.

The layer state is a private key $sk \in \mathsf{T}^{1024}$. The key sk is generated using prng and must be kept in secret. The corresponding public key pk, on the contrary, is publicly announced.

The layer contains the following algorithms:

- Gen (generate keys, 10.3);
- Encr (encrypt a session key, 10.4);
- Decr (decrypt a session key, 10.5).

10.2 Polynomials

Let $n = 1024$ and $q = 12289$.

An word $u = u[0]u[1] \ldots u[n-1]$ in an alphabet of integers is associated with the polynomial

$$u(x) = u[0] + u[1]x + \ldots + u[n-1]x^{n-1}$$

which degree is less than n. In turn, the word u can be reconstructed from a polynomial $u(x)$ by gathering its coeffients.

Having another such polynomial $v(x)$, one can calculate $u(x) \pm v(x)$ and $u(x)v(x)$ modulo $x^n + 1$. Due to the reduction, the degrees of the resulting polynomials remain below n.

A polynomial $u(x)$ can be also reduced mods 3 or q. The reduction is applied to each coefficient of $u(x)$ or, alternatively, to each symbol of u. Let $\operatorname{mods}(x^n + 1, 3)$ denote the reduction first modulo $x^n + 1$ and second modulo 3. The notation $\operatorname{mods}(x^n + 1, q)$ has a similar meaning.

Polynomials $\operatorname{mods}(x^n+1, 3)$ are naturally encoded by words from T^n. A code word consists of sequental coefficients $u[0]u[1] \ldots u[n-1]$ of an encoded polynomial $u(x)$.

Polynomials $\operatorname{mods}(x^n + 1, q)$ are encoded by words of T^{9n}. To encode a polynomial $u(x)$, its coefficients $u[0], u[1], \ldots, u[n-1]$ are interpreted as trints (it is important that $q < 27^3$) and then these trints are written from left to right as 9-trit blocks. To decode a word u, its 9-trit sequental blocks are interpreted as trints $u[0], u[1], \ldots, u[n-1]$ and then these trints are interpreted as coefficients of $u(x)$. If some coefficient $u[i]$ does not belong to the interval $\{-(q-1)/2, -(q-3)/2, \ldots, (q-1)/2\}$, then the decoding ends with an error.

Polynomials $\operatorname{mods}(x^n + 1, q)$ form a ring. This ring contains both invertible and non-invertible elements. If $u(x)$ is invertible, then there exists $v(x)$ such that

$$u(x)v(x) \operatorname{mods}(x^n + 1, q) = 1.$$

The polynomial $v(x)$ is called inverse of $u(x)$ and denoted as $(u(x))^{-1} \operatorname{mods}(x^n + 1, q)$.

10.3 `Gen`

Input: $N \in \mathsf{T}^*$ (a nonce).

Output: $pk \in \mathsf{T}^{9216}$ (a public key).

Steps:

1. $i \leftarrow 0$.
2. $r \leftarrow \texttt{prng.Gen}[\texttt{NTRUKEY}](N \parallel \langle i \rangle_{81}, 2048)$.
3. Represent r as $f \parallel g$ and reconstruct $f(x)$ and $g(x)$.
4. If either $1 + 3f(x)$ or $g(x)$ is not invertible $\operatorname{mods}(x^{1024} + 1, 12289)$, then:
   1) $i \leftarrow i + 1$;
   2) go to Step 2.
5. Encode $f(x)$ by $sk \in \mathsf{T}^{1024}$.
6. $h(x) \leftarrow 3g(x)(1 + 3f(x))^{-1} \operatorname{mods}(x^{1024} + 1, 12289)$;
7. Encode $h(x)$ by $pk \in \mathsf{T}^{9216}$.
8. Return pk.

10.4 `Encr`

Input: $K \in \mathsf{T}^{243}$ (a session key), $pk \in \mathsf{T}^{9216}$ (a public key), $N \in \mathsf{T}^*$ (a nonce).

Output: $Y \in \mathsf{T}^{9216}$ (an ecnrypted session key).

Steps:

1. $r \leftarrow \texttt{prng.Gen}[\texttt{NTRUKEY}](pk[\ldots 81) \parallel K \parallel N, 1024)$.
2. Decode pk to the polynomial $h(x) \operatorname{mods}(x^{1024} + 1, 12289)$.
3. $s(x) \leftarrow r(x)h(x) \operatorname{mods}(x^{1024} + 1, 12289)$.
4. Encode $s(x)$ by $s \in \mathsf{T}^{9216}$.
5. $\texttt{Spongos.Init}(\perp)$.
6. $\texttt{Spongos.Absorb}(s)$.
7. $\texttt{Spongos.Commit}(\perp)$.
8. $K \leftarrow \texttt{Spongos.Encr}(K)$.
9. $\texttt{Spongos.Commit}(\perp)$.
10. $t \leftarrow \texttt{Spongos.Squeeze}(1024 - 243)$.

11. $s(x) \leftarrow (s(x) + (K \parallel t)(x)) \operatorname{mods} 12289$.

12. Encode $s(x)$ by $Y \in \mathsf{T}^{9216}$.

13. Return Y.

Remark. A unique nonce N provides guarantees that a ciphertext Y for the same recipient varies even if K repeats. These guarantees are known in cryptography as semantic security. They could be useful if, for example, K is a non-volatile message which is sent twice to the same recipient. But in MAM2, K is a volatile session key and semantic security is usually redundant. So, it will not be a problem if $N = \perp$.

10.5 `Decr`

Input: $Y \in \mathsf{T}^{9216}$ (an encrypted session key), $sk \in \mathsf{T}^{1024}$ (a private key).

Output: K (a session key) or $\perp$ (a error).

Steps:

1. Decode sk to the polynomial $f(x) \operatorname{mods}(x^{1024} + 1, 3)$.

2. Decode Y to the polynomial $s(x) \operatorname{mods}(x^{1024} + 1, 12289)$. Return $\perp$ if a decoding error occurs.

3. $r(x) \leftarrow s(x)(1 + 3f(x)) \operatorname{mods}(x^{1024} + 1, 12289)$.

4. $r(x) \leftarrow r(x) \operatorname{mods} 3$.

5. $s(x) \leftarrow (s(x) - r(x)) \operatorname{mods} 12289$.

6. Represent r as $K \parallel t$, where $K \in \mathsf{T}^{243}$ and $t \in \mathsf{T}^{1024-243}$.

7. Encode $s(x)$ by $s \in \mathsf{T}^{9216}$.

8. `Spongos.Init`$(\perp)$.

9. `Spongos.Absorb`(s).

10. `Spongos.Commit`$(\perp)$.

11. $K \leftarrow$ `Spongos.Decr`(K).

12. `Spongos.Commit`$(\perp)$.

13. If $t \neq$ `Spongos.Squeeze`$(1024 - 243)$, then return $\perp$.

14. Return K.

10.6 Implementation issues

Multiplicative operations $\operatorname{mods}(x^n+1, q)$ are the heaviest component of the above algorithms. They can be sped up using several techniques. The most perspective approach is Number Theoretic Transform (NTT), a specialized version of Discrete Fourier Transform (DFT).

Let an integer γ have order $2n$ modulo q and let $\omega = \gamma^2 \operatorname{mods} q$ be the corresponding element of order n. For example, with $(n, q) = (1024, 12289)$ one can choose $\gamma = 7$ so that $\omega = 49$.

If a is coprime to q, then the multiplicative inverse $b = a^{-1} \operatorname{mods} q$ is defined: $ab \operatorname{mods} q = 1$. Negative powers $a^{-j} \operatorname{mods} q$ should be understood as $b^j \operatorname{mods} q$.

If $u(x) = u[0] + u[1]x \ldots + u[n-1]x^{n-1}$ is some polynomial $\operatorname{mods}(x^n+1, q)$, then $\mathrm{NTT}(u)$ is a polynomial $\hat{u}(x) = \hat{u}[0] + \hat{u}[1]x + \ldots + \hat{u}[n-1]x^{n-1}$ with the coefficients

$$\hat{u}[j] = \sum_{i=0}^{n-1} \gamma^i u[i]\omega^{ij} \operatorname{mods} q, \quad j = 0, 1, \ldots, n-1.$$

In other direction, $u = \mathrm{NTT}^{-1}(\hat{u})$ is a polynomial with the coefficients

$$u[i] = \left(n^{-1}\gamma^{-i} \sum_{j=0}^{n-1} \hat{u}[j]\omega^{-ij}\right) \operatorname{mods} q.$$

The following facts can be used to implement multiplicative operations $\operatorname{mods}(x^n+1, q)$ effectively.

1. The polynomial u is invertible if and only if all the coefficient of $\mathrm{NTT}(u)$ are nonzero.

2. If v is another polynomial, then

   $$uv = \mathrm{NTT}^{-1}(\mathrm{NTT}(u) \odot \mathrm{NTT}(v)),$$

   where $\odot$ is coefficient-wise multiplication of polynomials.

3. If v is invertible, then

   $$uv^{-1} = \mathrm{NTT}^{-1}(\mathrm{NTT}(u) \oplus \mathrm{NTT}(v)),$$

   where $\oplus$ is coefficient-wise division of polynomials.

4. $\mathrm{NTT}(u)$ and $\mathrm{NTT}^{-1}(u)$ can be calculated in $O(n \log n)$ operations $\operatorname{mods} q$ using the Fast Fourier Transform (FFT) technique. The choice of n as a power of 2 facilitates FFT.

11 The `Protobuf3` layer

11.1 Overview

The `Protobuf3` layer supports encoding, decoding and cryptographic processing of structured data. Processed data are described by a special data definition language called Protobuf3. This language is based on the well-known Protocol Buffers Version 2 notation (`https://developers.google.com/protocol-buffers/`).

The layer state consists of:

- a reference to an instance `mam2` of the `MAM2` layer;
- an instance `spongos` of the `Spongos` layer.

During encoding the layer makes calls to `mam2`. Through these calls, the final data structure and field values are determined. Actually, `mam2` pre-creates the target instance of `Protobuf3` and runs its algorithm `Encode` providing necessary data and settings on demand.

The same strategy (run-providing-data) is used during cryptographic processing: `Protobuf3` makes calls to `mam2` when some cryptographic fields (for example, signatures) have to be determined or verified. In turn, `mam2` translates these calls into the calls to its encapsulated cryptographic layers (for example, `MSS`).

The `Protobuf3` layer additionally refers to `spongos` for generation and verification of MACs, for encryption and decryption. The `spongos` instance can be forked during processing.

The `Protobuf3` layer contains the following algorithms:

- `Init` (initialize a state, 11.3);
- `Wrap` (encode structured data into a ternary stream, 11.4);
- `Unwrap` (decode from a ternary stream, 11.5).

11.2 The language

Building blocks of ProtoBuf3 are user-defined data types marked with the `message` keyword. Each type consists of fields. Each field has a name and a type. A field type can be either a base type, a composite type or a user-defined type.

Base types. Base types are the following:

- `null`: a special type that describes the absence of data;
- `tryte`: an element of T^3. Interpreted as an integer, takes values in the range $[-13, 13]$;
- `trint`: an element of T^9. Interpreted as an integer, consists of 3 trytes, takes values in the range $[-9841, 9841]$;
- `long trint`: an element of T^{18}. Interpreted as an integer, consists of 6 trytes, takes values in the range $[-193710244, 193710244]$.

Composite types. Composite types are the following:

- `trytes`: an array of trytes. The length of the array is implicitly encoded with the array elements;
- `T arr[n]`: an array `arr` of `n` elements of type `T`;
- `T arr[]`: an array `arr` of elements of type `T`. The array can be placed only at the end of a data object. Elements of `arr` are continued until the end of the object. The number of elements is not fixed during encoding, it is determined indirectly during decoding.

Modifiers. Fields are marked with the following modifiers:

– `oneof` — this field can be chosen from a given set of alternatives. The total number of alternatives must not exceed 27. Each alternative is marked with an integer from the set $\{-13, -12, \ldots, 13\}$. This integer is written (with the preceding sign `=`) in the ending of the field description line;

– `repeated` — this field can be repeated any number (including zero) of times.

The combination `repeated oneof` is possible but not the combination `oneof repeated`.

Cryptographic modifiers. Additional cryptographic modifiers control cryptographic data processing using `spongos`. These modifiers are:

– `absorb` — this field is absorbed;

– `squeeze` — this field is squeezed;

– `crypt` — this field is encrypted or decrypted;

– `skip` — this field must not be processed by `spongos`;

– `external` — this field is an external object. It can be absorbed or squeezed by `spongos` although not being presented in the resulting ternary stream.

These modifiers cannot be assigned to fields of user-defined types.

A field can have only one of the modifiers `absorb`, `squeeze`, `crypt` and `skip`. The `external` modifier can be combined with any of them.

If no one of the modifiers `absorb`, `crypt`, `squeeze` and `skip` is assigned to a field explicitly, then `absorb` is assigned implicitly.

Two additional modifiers control a state of `spongos`:

– `fork` — call $\texttt{spongos}' \leftarrow \texttt{spongos.Fork}(\perp)$. All fields after this call and until the end of the current user-defined type must be processed using $\texttt{spongos}'$. The `spongos` object should still be used to process the main message stream;

– `commit` — force $\texttt{spongos.Commit}(\perp)$. Usually used immediately after absorbing a key or nonce.

Actually, `fork` and `commit` are commands for `spongos`. But we assume for coherence that they are modifiers of empty (`null`) fields.

Encoding rules. Encoding rules are presented in Table 4.

In the table, `size_t` is an internal type used only for encoding. Values of `size_t` describe numbers of nested elements in such constructions as `trytes` and `repeated`.

Table 4: Encoding rules

type / modifier	code
message	The concatenation of codes of consecutive nested fields (recursively)
null	$\perp$
tryte	The corresponding word of $\mathbf{T}^3$
trint	The corresponding word of $\mathbf{T}^9$
long trint	The corresponding word of $\mathbf{T}^{18}$
size_t	A non-negative integer U of type size_t is encoded as $\langle U \rangle$
trytes	The code of the number of trytes (size_t) concatenated with these (consecutive) trytes
T arr[n]	The concatenation of the codes of arr[0], arr[1],..., arr[n-1]
T arr[]	The concatenation of the codes of arr[0], arr[1],...(until the encoded data object runs out)
oneof	The code (one tryte) of the chosen alternative
repeated	The code of the number of repetitions (size_t) concatenated with codes of these (consecutive) repetitions

11.3 Init

Input: mam2′ (a reference to an instance of MAM2), spongos′ (a reference to an instance of Spongos).

Output: $\perp$.

Steps:

1. mam2 $\leftarrow$ mam2′.
2. If spongos′ $=\perp$, then spongos.Init($\perp$), else spongos $\leftarrow$ spongos′.

11.4 Wrap

Input: T (a Protobuf3 type).

Output: $Y \in \mathbf{T}^{3*}$ (an encoded instance of the type).

Steps:

1. $Y \leftarrow \perp$.
2. Making calls to mam2, process fields of T:
    1) choose among alternatives in oneof fields;
    2) determine numbers of repetition of repeated fields;
    3) determine lengths of trytes fields;

4) determine values of `external` fields;

5) determine values of all other fields when it is possible to do without cryptographic processing.

3. Obtain, in result, a sequence of fields of base types. These fields have the unprocessed modifiers `fork`, `commit`, `absorb`, `squeeze`, `crypt`, `skip` and `external`.

4. Initialize an array $S[f]$ which indices are fields of the obtained sequence and which entries are references to instances of `Spongos`. Initially, all entries refer to `spongos`.

5. Process consecutive fields of the obtained sequence. For each field f:

   1) if f (of type `null`) has the `fork` modifier, then:

      (a) $\texttt{spongos}' \leftarrow S[f].\texttt{Fork}(\perp)$;

      (b) for all fields g from f up to the end of f's user-defined type: $S[g] \leftarrow \texttt{spongos}'$;

      (c) go to Step 10);

   2) if f (of type `null`) has the `commit` modifier, then:

      (a) $S[f].\texttt{Commit}(\perp)$;

      (b) go to Step 10);

   3) encode f by the rules of Table 4 and obtain a prefix $p \in \mathsf{T}^{3*}$ and a value $v \in \mathsf{T}^{3*}$. The possible non-empty prefixes are:

      – a code (one tryte) of the choice made in `oneof`;

      – a code (`size_t`) of the number of repetitions used in `repeated`;

      – a code (`size_t`) of the length of `trytes`.

      The value v is undefined if f has the modifier `squeeze` or `skip`. In any case, the length of v must be known at the moment;

   4) $S[f].\texttt{Absorb}(p)$;

   5) if f has the `absorb` modifier, then $S[f].\texttt{Absorb}(v)$;

   6) if f has the `squeeze` modifier, then $v \leftarrow S[f].\texttt{Squeeze}[|v|](\perp)$;

   7) if f has the `crypt` modifier, then $v \leftarrow \texttt{S}[f].\texttt{Encr}(v)$;

   8) if f has the `skip` modifier, then, making calls to `mam2`, recognize the semantic of the field. If f contains a signature, then:

      (a) recognize the previously squeezed field value v' to sign;

      (b) pass v' to `mss`, an appropriate instance of `MSS` encapsulated into `mam2`;

      (c) $v \leftarrow \texttt{mss.Sign}(v')$;

   9) if f does not have the `external` modifier, then $Y \leftarrow Y \parallel p \parallel v$;

   10) continue.

6. Return Y.

11.5 `Unwrap`

Input: T (a Protobuf3 type), $Y \in \mathbf{T}^{3*}$ (an encoded instance of the type).

Output: a decoded instance of the type (implicitly, through calls to `mam2`) or $\perp$.

Steps:

1. Run `Wrap` with the following corrections:

   1) provide settings for fields to `mam2` instead of getting them;
   2) provide field values to `mam2` instead of getting them;
   3) change `Wrap` recursive calls to `Unwrap` calls;
   4) verify MACs and signatures instead of generating them;
   5) process `crypt` fields using the `spongos.Decr` not `spongos.Encr` algorithm;
   6) return $\perp$ in the case of decoding or cryptographic errors.

12 The `MAM2` layer

12.1 Overview

The `MAM2` layer supports high-level operations of the MAM2 protocol. The main operations are sending and receiving messages.

The layer makes calls to the `MSS` and `Protobuf3` layers and to the global instance `prng` of the `PRNG` layer (see 7.1). The `prng` object must be pre-initialized.

The layer state consists of:

- a list of supported channels. Each channel is described by a name $N \in \mathbf{T}^*$ and an instance `mss` of `MSS`;
- a list of supported endpoints. Each endpoint is described by a name N of the outer channel, an own name $N' \in \mathbf{T}^*$ and an instance `mss` of `MSS`.

The `MAM2` layer contains the following algorithms:

- `CreateChannel` (create a channel, 12.3);
- `CreateEndpoint` (create an endpoint, 12.4);
- `Send` (send a message, 12.5);
- `Recieve` (recieve a message, 12.6).

To securely run these algorithms, the following additional key management services should be implemented:

1) authenticated distribution of channel identifiers (actually, public keys);
2) authenticated and confidential distribution of preshared keys;

3) authenticated distribution of public keys.

These services are mostly outside the scope of this specification.

Warning. A channel owner must not use the same channel name twice. Duplication of names causes the repetition of the keys of the encapsulated `mss` objects which, in turn, makes the generated signatures unsafe. For the same reason, the channel owner must avoid duplication of endpoint names within a channel.

Warning. A channel owner should avoid transferring the encapsulated `mss` objects between several devices. Such a tranfer may cause the same key to be used twice in parallel. This also makes the generated signatures unsafe.

12.2 The format

A MAM2 message is described by the following ProtoBuf3 type:

```
message Msg {
  Channel channel;
  Endpoint endpoint;
  Header header;
  Packet packets[];
}
```

The fields of `Msg` have the following meaning:

- `channel` — a description of a channel to which the message belongs;
- `endpoint` — a description of an endpoint which is used to transmit the message. If this field is absent then the central channel endpoint (equipped with `chid`) is used;
- `header` — a header of the message that contains keyloads for different recipients or groups of recipients. Using some keyload, a recipient can recover a session key K which has been used to protect the message;
- `packets` — message packets.

The `Channel` type. A MAM2 channel is described by the following type:

```
message Channel {
  tryte ver;
  external tryte chid[81];
}
```

The fields of `Channel` have the following meaning:

- `ver` — a version of MAM2. The current version is 0;
- `chid` — an identifier of the channel. The `external` modifier says that the identifier isn't actually presented in the `Channel` container: it is put in an outer transport container, for example, in the `address` field of an outer IOTA bundle (see Chapter 14).

The `Endpoint` type. A MAM2 endpoint is described by the following type:

```
message Endpoint {
  oneof pubkey {
    null chid = 0;
    tryte epid[81] = 1;
    SignedId chid1 = 2;
    SignedId epid1 = 3;
  }
}
message SignedId {
  tryte id[81];
  MSSig mssig;
}
message MSSig {
  commit;
  external squeeze tryte mac[78];
  skip trytes sig;
}
```

The `pubkey` field of the `Endpoint` type describes a public key that can be used to verify signatures of endpoint messages or their parts. It could be either:

– `chid` — a key of this channel;

– `chid1` — a key of another channel;

– `epid` and `epid1` — a subordinate endpoint key.

In the `chid1` and `epid1` cases, a public key is accompanied with a signature. A signed public key is described by the `SignedId` type, where `id` is the key itself and `mssig` is its signature.

The signature must be generated using the `MSS` layer with the private key that corresponds to `chid`. Signing a public key in `chid1` (`epid1`), an owner of the `chid` channel authenticates the announcement of a new channel (endpoint).

The unsigned `epid` key must be signed somewhere earlier, that is, it must be published in the `epid1` field of some previous message in this channel.

In the `MSSig` type, the `sig` field must contain a signature of the `mac` field.

The `Header` type. A message header is described by the following types:

```
message Header {
  tryte msgid[21];
  trint typeid;
  repeated oneof keyload {
    KeyloadPSK psk = 1;
    KeyloadNTRU ntru = 2;
  }
  external tryte key[81];
```

```
    commit;
}

message KeyloadPSK {
    fork;
    tryte id[27];
    external tryte psk[81];
    commit;
    crypt tryte ekey[81];
}
message KeyloadNTRU {
    fork;
    tryte id[27];
    tryte ekey[3072];
}
```

The fields of `Header` have the following meaning:

- `msgid` — a unique message nonce which stands as a message identifier;
- `typeid` — an identifier of the message type. Specifying the type makes it easier to parse the message payload. Type identifiers for standard messages are defined in Chapter 13 along with the messages themselves. The zero identifier is reserved for messages with unstructured payloads;
- `keyload` — a tuple of keyloads of different types. If this field is empty (no keyloads are presented), then the message is transmitted in public mode. Its payload can be decrypted by anyone although integrity and authenticity control is still possible;
- `key` — a secret session key K that is implicitly inserted in the message stream. The insertion makes a sponge state secret and allows to generate MACs and produce ciphertexts in subsequent packets. If the previous field is empty (no keyloads), then K must be reset to zero. Resetting the key makes it public and thus turns on public mode.

Each keyload starts with the `fork` modifier. It means that cryptographic processing of keyload fields is performed in a branch of the main message stream. In particular, MACs or signatures of the main stream do not control the keyload data.

The `KeyloadPSK` type describes the PSK (Pre-Shared Key) keyload. This keyload contains a session key encrypted using a previously delivered PSK. The `id` field presents an identifier of a group of recipients who share the same PSK key. The `psk` field presents this key, and the `ekey` field presents the encrypted K.

The `KeyloadNTRU` type describes the NTRU keyload. This keyload contains a session key encrypted using recipient's public key. The `NTRU` layer and, more precisely, the `NTRU.Encr` algorithm is used for encryption. The `id` field of `KeyloadNTRU` presents first trytes of recipient's public key and the `ekey` field presents the encrypted K. The trusted delivery of recipient's public keys can be done using public key certificates defined in 13.2.

The `Packet` type. A message packet is described by the following type:

```
message Packet {
  long trint ord;
  crypt trytes payload;
  oneof checksum {
    null none = 0;
    MAC mac = 1;
    MSSig mssig = 2;
  }
  commit;
}
message MAC {
  commit;
  squeeze tryte mac[81];
}
```

The fields of `Packet` have the following meaning:

- `ord` — an ordinal number of the packet. Packets are numbered from one, the number of the last packet is made negative. For example, if a message contains 3 packets, then their numbers are $1, 2, -3$, and if a message contains a single packet, then its number is -1;
- `payload` — an informative part of the message. It is encrypted using a session key K;
- `checksum` — some control characteristic of the current packet data as well as previous packets and message headers. It could be a MAC (the `mac` field), a signature of the MAC (`mssig`) or the empty field (`none`).

Note that in public mode a MAC becomes a keyless hash value which doesn't provide integrity and authenticity control. To provide such control, a MAC must be signed, that is, the `mssig` field must be chosen in `checksum`.

12.3 `CreateChannel`

Input: d (a height: 2^d channels / endpoints / messages can be signed), $N \in \mathbf{T}^{3*}$ (a channel name).

Output: `chid` $\in \mathbf{T}^{243}$ (an identifier of the created channel) or $\perp$.

Steps:

1. If a channel with the name N is already created, then return $\perp$.
2. Create and save in the state an instance `mss` of the `MSS` layer along with the channel name N.
3. `chid` $\leftarrow$ `mss.Gen`$(d, \langle |N|/3 \rangle \parallel N)$.
4. Return `chid`.

12.4 `CreateEndpoint`

Input: d (a height: 2^d messages can be signed), $N \in \mathsf{T}^{3*}$ (a channel name), $N' \in \mathsf{T}^{3*}$ (an endpoint name).

Output: $\mathtt{epid} \in \mathsf{T}^{243}$ (an identifier of the created endpoint) or $\bot$.

Steps:

1. If a channel with the name N doesn't exist, then return $\bot$.
2. If the channel N already contains an endpoint with the name N', then return $\bot$.
3. Create and save in the state an instance `mss` of the `MSS` layer along with the channel name N and the endpoint name N'.
4. $\mathtt{epid} \leftarrow \mathtt{mss.Gen}(d, \langle |N|/3 \rangle \parallel N \parallel \langle |N'|/3 \rangle \parallel N')$.
5. Return `epid`.

12.5 `Send`

Input: $X \in \mathsf{T}^{3*}$ (a plain message).

Output: $Y \in \mathsf{T}^{3*}$ (a protected message ready to send on wire).

Steps:

1. Select a channel `chid` in which X will be sent. Let N be the name of the chosen channel.
2. Select an endpoint `epid` which will be used to transmit X. Let N' be the name of the selected endpoint. The central endpoint (that is, the channel itself) can be selected. In this case, N' is undefined.
3. Determine `mss`, an instance of `MSS` which corresponds to the chosen endpoint.
4. Split X into parts (packets) which are strings of trytes. These strings will be placed at the `payload` field of the `Packet` type. Details of splitting are beyond the scope of this specification.
5. Choose msgid $\in \mathsf{T}^{81}$, an identifier of the current message. The identifier must be unique within the chosen endpoint.
6. Generate a session key K:
   1) $K \leftarrow \mathtt{prng.Gen}[\mathtt{SECKEY}](\langle |N|/3 \rangle \parallel N \parallel \mathtt{msgid}, 243)$ if the central endpoint was selected at Step 2;
   2) $K \leftarrow \mathtt{prng.Gen}[\mathtt{SECKEY}](\langle |N|/3 \rangle \parallel N \parallel \langle |N'|/3 \rangle \parallel N' \parallel \mathtt{msgid}, 243)$ if a regular endpoint N' was selected at Step 2.
7. Choose recipients and keyloads for them.
8. Reset K to zero (activate public mode) if no keyloads were chosen.

9. Create a temporary instance `pb` of the `Protobuf3` layer.

10. Run $\texttt{pb.Init}(\texttt{this}, \perp)$, where `this` is a reference to the current instance of `MAM2`.

11. Determine $Y \leftarrow \texttt{pb.Wrap}(\texttt{Msg})$ providing `pb` with the required information inluding X's packets.

12. Return Y.

12.6 Recieve

Input: $Y \in \mathsf{T}^{3*}$ (a protected message).

Output: $X \in \mathsf{T}^{3*}$ (a plain message) or $\perp$.

Steps:

1. Create a temporary instance `pb` of the `Protobuf3` layer.

2. Run $\texttt{pb.Init}(\texttt{this}, \perp)$, where `this` is a reference to the curent instance of `MAM2`.

3. Run $\texttt{pb.Unwrap}(\texttt{Msg}, Y)$ providing `pb` with the required information.

4. If `Unwrap` outputs $\perp$, then repeat this output. Otherwise, `Unwrap` provides decrypted authenticated packets. Gather these packets into X.

5. Return X.

13 Messages

13.1 Overview

In this section, we standardize useful messages that can be distributed via MAM2 channels. The list of standard messages is open, it may be updated in the future versions of this specification.

Messages are defined using the Protobuf3 language. Before transmitting, they are encoded using Protobuf3 rules and represented as strings of trytes. These strings are divided into fragments which are put into MAM2 packets. Rules of fragmentation are beyond the scope of this specification.

Note that during the encoding messages are not processed cryptographically (this will be done later, in the `MAM2` layer), so the cryptographic modifiers must not present in Protobuf3 descriptions of the messages.

Type identifiers of standard messages are presented in Table 5. These identifiers should be presented in the `Header.typeid` field (see 12.2) of messages.

Table 5: Type identifiers

message	identifier
Unstructured*	0
Public key certificate (see 13.2)	1

* or a sender doesn't want to reveal a structure

13.2 Public key certificates

A public key certificate (PKC) represents a public key and its owner. Publishing a certificate in a channel in the signed form, the owner of the channel approves the validity of the certificate's data. In this way, MAM2 channels can serve as trusted sources of public keys and provide PKI (Public Key Infrastructure) for IOTA/MAM2.

A certificate is described by the following ProtoBuf3 type:

```
message Cert {
  trytes name;
  oneof pubkey {
    tryte chid[81] = 1;
    tryte ntru[3072] = 2;
}
```

The fields of `Cert` have the following meaning:

- `name` — a description of a certificate owner. A format of this description is beyond the scope of this specification. The string `name` can be empty if an anonymous key-centric PKE is intended;

- `pubkey` — a description of a public key. It can be either a public key of another channel (the `chid` option) or a public key of the `NTRU` layer (the `ntru` option).

14 Transport over the Tangle

The default transport layer for MAM2 messages is the Tangle, a distributed database used in the IOTA infrastructure. For transmission over the Tangle, MAM2 messages are split into small fragments called transactions. The sequence of transactions is divided into blocks called bundles. Exactly bundles are records of the Tangle database.

A bundle can contain either:

a) a header of a message;

b) a header of a message and its first packet;

c) a standalone message packet.

In case b), the first bundle's transactions transfer the header and the last transactions transfer the first message packet. There are no transactions that contain both a header and a packet. Therefore, transactions are divided into two categories: header transactions and packet transactions.

In addition to the encapsulated MAM2 data, bundle's transactions contain metatada. The following metadata fields are related to the MAM2 transport:

```
tryte address[81];
tryte tag[27];
```

The `address` field must contain an identifier of the channel (see `Channel.chid` in 12.2).

The `tag` field must be a concatenation of two strings:

1) an identifier of the message (`Header.msgid`);

2) a string of 6 trytes (18 trits) which represents either

   – the zero number in header transactions or

   – the ordinal number of a packet (`Packet.ord`) in packet transactions.

   The representation is built using the standard operation $\langle\cdot\rangle_{18}$.

Note that the ordinal number `Packet.ord` is made negative for the last message packet. So to find the packet number n in a message `msgid`, one needs to test two options in `tag`: `msgid` $\|\ \langle n\rangle_{18}$ and `msgid` $\|\ \langle -n\rangle_{18}$. The last option corresponds to the last packet.

To save space and avoid data duplication, when `Header` is placed in header transactions, its `msgid` field is omitted. In the other direction, when `Header` is output from header transactions, its `msgid` field is restored from the `tag` field of transactions. Similarly, `Packet.ord` is excluded from packet transactions and restored from their `tag` field.