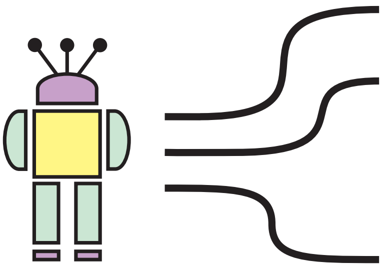
Everything is a message

MAM1.1 (Masked Authenticated Messaging Version 1.1) Protocol



Revision: 0.8
Date: 2019.11.14

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 9.4) Insert an additional commit in `NTRU.Encr/Decr` (see 9.4, 9.5) Rename `Header.nonce` to `Header.msgid` Add `Header.typeid` Remove `KeyloadPlain` from `Header.keyload` Make `Header.ord` negative in the last packet Add Chapter ?? (Transport over the Tangle)
0.6	2019.02.18	Add $\langle\cdot\rangle$ notation which supports `size_t` (see 1, 10.2) Warn about duplication / parallel usage of names / keys 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
0.7	2019.02.27	Shorten `Header.msgid` to 21 trytes Change semantics of the `tag` field (see ??)

0.8	2019.11.14	Updated Preliminaries section (see 4)
		Removed `Sponge` layer
		Pass `spongos` explicitly in `NTRU.Encr/Decr` (see 9.4, 9.5)
		Added `Spongos Join` operation (see 5.10)
		Reimplemented `PRNG` using `Spongos` (see 6.3)
		Updated Protobuf3 overview (see 10.1)
		Added PB3 `link` base type (see 10.2)
		Added PB3 `join`, `mssig`, `ntrukem` commands (see 10.2, 10.3, 10.4)
		Removed PB3 `Init`
		Removed `MAM2` layer
		Added `Application` layer (see 11)
		Changed message version to 1
		Removed `Messages` section in favour of `Channel` section
		Added `Channel` application (see 12)
		Added `Channel` application use-cases (see 12.9)

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 `Alg` with inputs $a_1, a_2, \ldots$;
$\mathtt{Alg}[p](a_1, a_2, \ldots)$	calling an algorithm `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;
$\|u\|$	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 $w \in \mathsf{T}^{\|u\|+\|v\|}$ such that $w[\ldots \|u\|) = u$ and $w[\|u\| \ldots) = v$;
$u \oplus v$	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

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

3 Glossary

3.1 trit: an element of $\mathbf{T}$;
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`	$00\bar{1}$	$\bar{9}$	`R`
100	1	`A`	101	10	`J`	$10\bar{1}$	$\bar{8}$	`S`
$\bar{1}10$	2	`B`	$\bar{1}11$	11	`K`	$\bar{1}1\bar{1}$	$\bar{7}$	`T`
010	3	`C`	011	12	`L`	$01\bar{1}$	$\bar{6}$	`U`
110	4	`D`	111	13	`M`	$11\bar{1}$	$\bar{5}$	`V`
$\bar{1}\bar{1}1$	5	`E`	$\bar{1}\bar{1}\bar{1}$	$\overline{13}$	`N`	$\bar{1}\bar{1}0$	$\bar{4}$	`W`
$0\bar{1}1$	6	`F`	$0\bar{1}\bar{1}$	$\overline{12}$	`O`	$0\bar{1}0$	$\bar{3}$	`X`
$1\bar{1}1$	7	`G`	$1\bar{1}\bar{1}$	$\overline{11}$	`P`	$1\bar{1}0$	$\bar{2}$	`Y`
$\bar{1}01$	8	`H`	$\bar{1}0\bar{1}$	$\overline{10}$	`Q`	$\bar{1}00$	$\bar{1}$	`Z`

3.4 application: a message-oriented cryptographic protocol;
3.5 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);
3.6 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 MAM1.1 (Masked Authenticated Messaging version 1.1) cryptographic protocol framework designated for the IOTA platform. The MAM framework introduces cryptographic primitives and data description languague allowing users to specify and implement cryptographic protocols. Such protocol is called MAM Application and defines interacting parties and their roles, protocol messages and their semantics.

This specification defines `Channel` Application. The channel Author can announce and change his long-term keys, provide session key material for a set of Subscribers and publish signed messages. Subscribers can request Author for subscription and unsubscription, fetch and verify signed by Author messages. Author and Subscribers can publish and fetch authenticated messages that cannot be authored.

Messages of `Channel` Application form a tree-like topology (see Figure 1) allowing for various use-cases discussed in more detail in 12.9. For comparison, message topologies of MAM0 and MAM1 are shown in Figures 2 and 3 correspondingly.

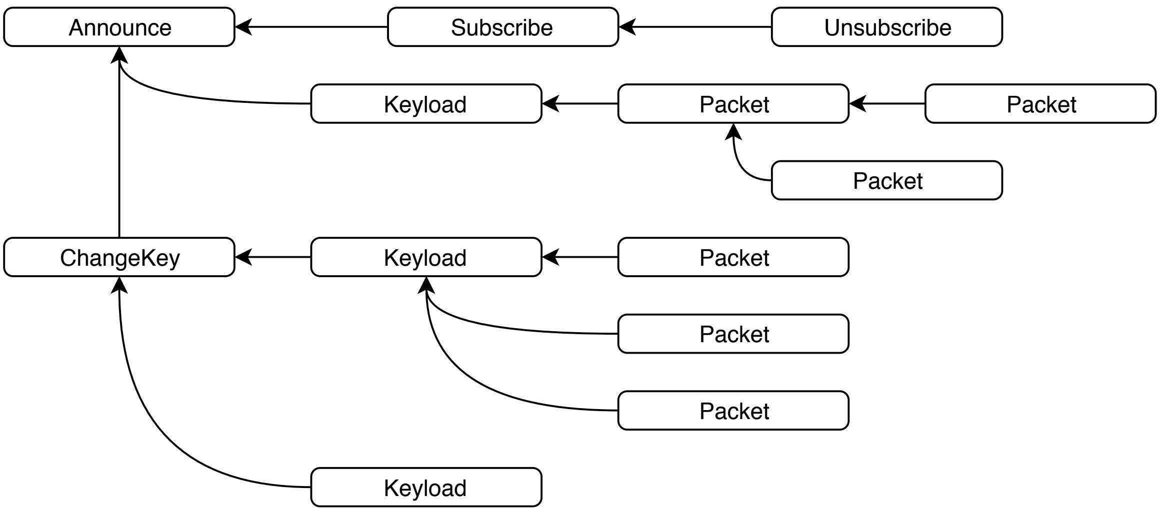


Figure 1: Message topology of `Channel` Application

Figure 2: Message topology of MAM0

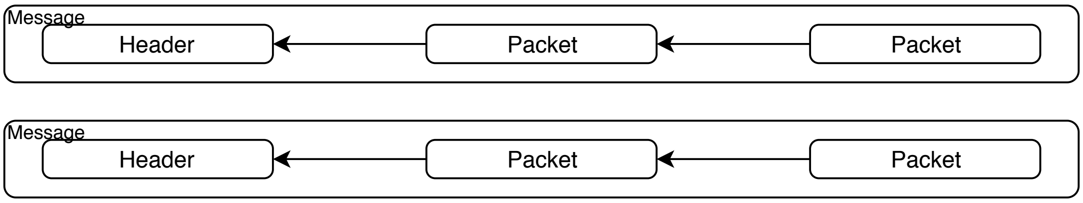


Figure 3: Message topology of MAM1

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

- `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);
- `Application` (specific application protocols).

The `Spongos` (from $\sigma\pi o\gamma\gamma o\varsigma$, "sponge" in Greek) is essentially a "language" for cryptographic data processing using F. `Spongos` layer is intended for working with strictly defined formats: we know exactly both the way of processing and the field length.

5 The `Spongos` layer

5.1 Overview

The `Spongos` layer supports operations based on a sponge function $F\colon \mathbf{T}^{729} \to \mathbf{T}^{729}$. 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 $S \in \mathbf{T}^{729}$ 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, 5.2);

- `Fork` (create an equivalent instance, 5.3);
- `Commit` (commit changes in the rate part, 5.4);
- `Absorb` (process input data, 5.5);
- `Squeeze` (generate output data, 5.6);
- `Hash` (hashing, 5.7);
- `Encr` (encrypt plaintext, 5.8);
- `Decr` (decrypt ciphertext, 5.9).
- `Join` (join states, 5.10).

5.2 `Init`

Input: $\perp$.

Output: $\perp$.

Steps:

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

5.3 `Fork`

Input: $\perp$.

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

Steps:

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

5.4 `Commit`

Input: $\perp$.

Output: $\perp$.

Steps:

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

5.5 `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`$(\perp)$.

5.6 `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.

5.7 `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.

5.8 `Encr`

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

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

Steps:

1. $Y \leftarrow 0^{|X|}$.
2. For $i = 0, 1, \dots, |X| - 1$:
   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 $\texttt{Commit}(\perp)$.
3. Return Y.

5.9 `Decr`

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

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

Steps:

1. $X \leftarrow 0^{|Y|}$.
2. For $i = 0, 1, \dots, |X| - 1$:
   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 $\texttt{Commit}(\perp)$.
3. Return X.

5.10 `Join`

Input: $\texttt{spongos}'$ (joinee instance of `Spongos`).

Output: $\perp$.

Steps:

1. $\texttt{spongos}'.\texttt{Commit}(\perp)$.
2. $X \leftarrow \texttt{spongos}'.\texttt{Squeeze}(243)$.
3. $\texttt{Absorb}(X)$.
4. $\texttt{Commit}(\perp)$.

6 The PRNG layer

6.1 Overview

The PRNG layer supports the generation of cryptographically strong pseudorandom numbers or, more precisely, strings of trytes. The layer makes calls to the Spongos 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 MAM 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 2.

Table 2: 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, 6.2);

– Gen (generate pseudorandom numbers, 6.3).

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).

6.2 Init

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

Output: $\perp$.

Steps:

1. $K \leftarrow X$.

6.3 Gen

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

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

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

Steps:

1. $\texttt{Spongos.Init}(\perp)$.
2. $\texttt{Spongos.Absorb}(K \parallel d \parallel \langle |N| \rangle \parallel N \parallel \langle n \rangle)$.
3. $\texttt{Spongos.Commit}(\perp)$.
4. $Y \leftarrow \texttt{Spongos.Squeeze}(n)$.
5. Return Y.

7 The `WOTS` layer

7.1 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 6.1). The `prng` must be pre-initialized.

The layer state is a private key $sk \in \mathbf{T}^{13122}$. 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, 7.2);

– `Sign` (generate a signature, 7.3);

– `Recover` (recover a presumed public key from a signature, 7.4);

– `Verify` (verify a signature, 7.5).

7.2 `Gen`

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

Output: $pk \in \mathbf{T}^{243}$ (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.

7.3 `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.

7.4 `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.

7.5 `Verify`

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

8 The `MSS` layer

8.1 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 7.1). 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, 8.2);
- `Skn` (return d and skn, 8.3);
- `APath` (build an authentication path, 8.4);
- `Sign` (generate a signature, 8.5);
- `Verify` (verify a signature, 8.6).

8.2 `Gen`

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

Output: $pk \in \mathbf{T}^{243}$ (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 $mt[0,0]$.

8.3 `Skn`

Input: $\perp$.

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

Steps:

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

8.4 `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.

8.5 `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. $S \leftarrow$ `Skn`$(\perp)$.
3. $S \leftarrow S \parallel$ `wots`$[skn]$.`Sign`(H).
4. $S \leftarrow S \parallel$ `APath`(skn).
5. $skn \leftarrow skn + 1$.
6. Return S.

8.6 `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.

9 The `NTRU` layer

9.1 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 6.1). 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, 9.3);
- `Encr` (encrypt a session key, 9.4);
- `Decr` (decrypt a session key, 9.5).

9.2 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 $\mathbf{T}^n$. 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 $\mathbf{T}^{9n}$. 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 $(u(x))^{-1} \operatorname{mods}(x^n + 1, q)$.

9.3 `Gen`

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

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

Steps:

1. $i \leftarrow 0$.
2. $r \leftarrow$ `prng.Gen[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 $sk \in \mathbf{T}^{1024}$.

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.

9.4 `Encr`

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

Output: `spongos` (an updated `Spongos` instance), $Y \in \mathsf{T}^{9216}$ (an ecnrypted session key).

Steps:

1. $r \leftarrow$ `prng.Gen`[`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. `spongos.Absorb`(s).
6. `spongos.Commit`$(\perp)$.
7. $K \leftarrow$ `spongos.Encr`(K).
8. `spongos.Commit`$(\perp)$.
9. $t \leftarrow$ `spongos.Squeeze`$(1024 - 243)$.
10. $s(x) \leftarrow (s(x) + (K \parallel t)(x)) \operatorname{mods} 12289$.
11. Encode $s(x)$ by $Y \in \mathsf{T}^{9216}$.
12. Return `spongos`$, Y$.

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 MAM, K is a volatile session key and semantic security is usually redundant. So, it will not be a problem if $N = \perp$.

9.5 `Decr`

Input: `spongos` (a `Spongos` instance), $Y \in \mathbf{T}^{9216}$ (an encrypted session key), $sk \in \mathbf{T}^{1024}$ (a private key).

Output: `spongos` (an updated `Spongos` instance), 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 $K \in \mathbf{T}^{243}$ and $t \in \mathbf{T}^{1024-243}$.
7. Encode $s(x)$ by $s \in \mathbf{T}^{9216}$.
8. `spongos.Absorb`(s).
9. `spongos.Commit`$(\perp)$.
10. $K \leftarrow$ `spongos.Decr`(K).
11. `spongos.Commit`$(\perp)$.
12. If $t \neq$ `spongos.Squeeze`$(1024-243)$, then return $\perp$.
13. Return `spongos`, K.

9.6 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

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

   where $\oplus\!\!\!\!\div$ 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.

10 The `Protobuf3` layer

10.1 Overview

The `Protobuf3` layer supports encoding, decoding and cryptographic processing of formatted data. The data format and the data processing rules are described by the data definition language Protobuf3 (PB3 for short). This language takes after the well-known Protocol Buffers Version 2 notation (`https://developers.google.com/protocol-buffers/`).

The `Protobuf3` layer provides the following operations:

- `Wrap` (encode and process formatted data into a ternary stream, 10.3);

- `Unwrap` (decode and process formatted data from a ternary stream, 10.4).

These operations take PB3 data format description as input and run cryptoprocessing commands over data fields.

Commands may have different semantics such as signature generation and verification during `Wrap` and `Unwrap` operations. Commands may take different additional input arguments such as private and public keys during `Wrap` and `Unwrap` operations. The input data fields and additional arguments are provided by an Application instance `app` on demand.

`Spongos` commands implicitly use `spongos` instance associated with the current PB3 block. `fork` command forks the `spongos` instance and sets it as implicit for the subsequent commands in the current PB3 block. The forked `spongos` instance is destroyed at the end of the PB3 block.

10.2 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`: a signed integer type with values in the range $[-13, 13]$, encoded as an element of $\mathbf{T}^3$;
- `trint`: a signed integer type with values in the range $[-9841, 9841]$, consists of 3 trytes, encoded as an element of $\mathbf{T}^9$;
- `long trint`: a signed integer type with values in the range $[-193710244, 193710244]$, consists of 6 trytes, encoded as an element of $\mathbf{T}^{18}$;
- `link`: a link (reference) to another message; ternary representation depends on transport layer. `link` is represented as either `tryte[27]` when the Tangle is used for transport or `trytes` for other transports.

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 commands. Cryptographic commands control cryptographic data processing. The behaviour of the cryptographic commands may differ during `Wrap` and `Unwrap` operations. The following commands are field modifiers and operate upon the field trit code word:

- `absorb` — the field is absorbed;
- `squeeze` — the field is squeezed;

- mask — the field is masked, ie. encrypted during Wrap operation and decrypted during Unwrap;
- skip — the field is not processed by spongos;
- join — the spongos state of the message referenced by the field of link type is joined into the spongos;
- mssig(hash) — the field contains a MSS signature, the signature of a hash-value contained in the field hash of type tryte[78] is generated during Wrap operation and verified during Unwrap;
- ntrukem(key) — the field contains an NTRU encapsulation token of a key contained in the field key of type tryte[81], the key is encrypted during Wrap operation and decrypted during Unwrap. NTRU encapsulation token has it's own integrity check mechanisms and thus do not need to be absorbed by spongos instance.

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

A field can have only one of the modifiers absorb, squeeze, mask, skip and join.

If a field is assigned none of the cryptographic modifiers explicitly, then absorb is assigned implicitly.

Two additional commands control a state of spongos and do not have associated fields:

- fork — call spongos$'$ $\leftarrow$ spongos.Fork($\perp$). All fields after this call and until the end of the current user-defined type must be processed using spongos$'$. The spongos object should still be used to process the main message stream;
- commit — force spongos.Commit($\perp$). Usually used immediately after absorbing a key or nonce.

Storage modifier. external — the field is an external object processed cryptographically but not present in the resulting ternary stream.

external is usually used with absorb and squeeze commands but can be combined with any of the cryptographic modifiers.

Encoding rules. Encoding rules are presented in Table 3.

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 3: 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
`link`	The code of the value of the type corresponding to `link`
`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

10.3 `Wrap`

Input: `app` (a reference to an Application instance), T (a Protobuf3 type).

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

Steps:

1. `spongos.Init`($\perp$).
2. $Y \leftarrow \perp$.
3. Making calls to `app`, 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.
4. Obtain, in result, a sequence of fields of base types. These fields have the unprocessed modifiers `fork`, `commit`, `absorb`, `squeeze`, `mask`, `skip` and `external`.
5. 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`.

6. 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 13);
    2) if f (of type `null`) has the `commit` modifier, then:
        (a) $S[f].\texttt{Commit}(\perp)$;
        (b) go to Step 13);
    3) encode f by the rules of Table 3 and obtain a prefix $p \in \mathbf{T}^{3*}$ and a value $v \in \mathbf{T}^{3*}$. 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 `mask` modifier, then $v \leftarrow \texttt{S}[f].\texttt{Encr}(v)$;
    8) if f has the `skip` modifier, then $v \leftarrow f$;
    9) if f has the `mssig(hash)` modifier, then:
        (a) query `app` for `MSS` instance `mss`;
        (b) $H \leftarrow$ code of `hash` field;
        (c) $v \leftarrow \texttt{mss.Sign}(H)$;
    10) if f has the `ntru(key)` modifier, then:
        (a) query `app` for `NTRU` public key `pk` and nonce N;
        (b) $K \leftarrow$ code of `key` field;
        (c) $v \leftarrow \texttt{NTRU.Encr}(K, pk, N)$;
    11) if f (of type `link`) has the `join` modifier, then:
        (a) query `app` for `Spongos` instance $\texttt{spongos}'$ being the resulting state of processing the message referenced by f;
        (b) $S[f].\texttt{Join}(\texttt{spongos}')$;
        (c) $v \leftarrow f$;
    12) if f does not have the `external` modifier, then $Y \leftarrow Y \parallel p \parallel v$;
    13) continue.
7. Return Y.

10.4 Unwrap

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

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

Steps:

1. Run Wrap with the following corrections:

    1) provide settings for fields to app instead of getting them;
    2) provide field values to app instead of getting them;
    3) change Wrap recursive calls to Unwrap calls;
    4) verify MACs instead of generating them;
    5) process mask fields using the spongos.Decr not spongos.Encr algorithm;
    6) process mssig fields using the MSS.Verify with MSS public key pk provided by app;
    7) process ntrukem fields using the NTRU.Decr with NTRU private key sk provided by app;
    8) return $\perp$ in the case of decoding or cryptographic errors.

11 The Application layer

11.1 Overview

The Application layer introduces the notion of MAM Application. A MAM Application defines and implements a cryptographic protocol. A MAM Application is described by PB3 formats of it's messages, roles of interacting parties, their states, including own secret keys and trusted public keys, their actions during Wrap and Unwrap operations over messages and other application-specific logic.

11.2 Application base

All application messages have the following format:

```
message Message {
  Header header;
  Content content;
  commit;
}
message Header {
  absorb tryte version;
  absorb external tryte appinst[81];
  absorb external tryte msgid[27];
  absorb trytes type;
}
```

`Message` consists of `Header` and `Content`.

The fields of `Header` have the following meaning:

- `version` — a version of MAM1.1. The current version is 1;
- `appinst` — MAM Application instance identifier. It is stored externally in `address` field of IOTA Transaction;
- `msgid` — message identifier within the current application instance. It is stored externally in `tag` field of IOTA Transaction;
- `type` — an application-specifig string identifying the type of the `content` field. These identifiers must be unique among all Applications.

The `spongos` state after the `commit` command can be used as joinee in `join` command with link referencing this message, ie. link target equals `msgid`. Implementations of Applications may wish to cache these `spongos` states after all or some messages depending on Application instance needs.

12 The `Channel` MAM Application

12.1 Overview

The `Channel` MAM Application extends capabilities of MAM0 and MAM1.

The following are `Parties` of `Channel` Application:

- `Author` — `Channel` Application instance owner, maintains a set of MSS private keys and optionally an NTRU private key and a set of pre-shared with Subscribers secret keys; Author publishes signed keyloads and signed packets, handles subscription of Subscribers.
- `Subscriber` — `Channel` Application instance user, maintains optionally a pre-shared with Author secret key or an NTRU private key, can request Author for subscription and unsubscription, fetches and verifies signed keyloads and signed packets, publishes and fetches tagged packets.

`Channel` Application uses the following `Messages`:

- `Announce` — published by Author, contains his MSS public key and optionally NTRU public key, signed with corresponding MSS private key. It's the first message of the channel and announces creation of the channel.
- `ChangeKey` — published by Author, linked to an `Announce` message or to another `ChangeKey` message, contains Author's MSS public key, signed with corresponding MSS private key and with MSS private key corresponding to public key contained in the linked message. The former signature is a proof on knowledge of private key and the latter signature transfers trust of the linked public key to trust of the current public key.
- `Keyload` — published by either Author or Subscriber, linked to any channel instance message, contains nonce and secret session key encrypted for a set of recipients.

- `Subscribe` — published by a Subscriber, linked to an `Announce` message, contains Subscriber's NTRU public key and a secret "unsubscribe" key encapsulated with Author's NTRU public key, allows Subscriber to be included as a recipient in signed keyloads published by Author.

- `Unsubscribe` — published by a Subscriber, linked to corresponding `Subscribe` message, contains Subscriber's NTRU public key and a secret "unsubscribe" key encapsulated with Author's NTRU public key, lets Author to exclude Subscriber from the list of recipients in signed keyloads.

- `SignedPacket` — published by Author, linked to any message in the channel application instance, contains public and masked payloads signed with one of the Author's MSS private keys.

- `TaggedPacket` — published by either Author or Subscriber, linked to any message in the channel application instance, contains public and masked payloads, authenticated with MAC, thus hiding true identity of the message publisher.

Note, that masked payloads of `SignedPacket` and `TaggedPacket` messages are truly masked only when the linked `spongos` state is secret, ie. it has absorbed a symmetric key from a `Keyload` message.

In order to guarantee security of the `Channel` Application instance, the following additional key management services should be implemented:

1) authenticated distribution of Author's MSS public key published in `Announce` message;

2) authenticated distribution of Subscriber's NTRU public key published in `Subscribe` message;

3) authenticated and confidential distribution of preshared keys.

These services usually require a secure communication channel and are mostly outside the scope of this specification.

In order to simplify key management implementations may choose not to store secret keys individually but rather generate them using one global `prng` instance. In such case, in order to avoid possible key duplications among different Application instances, implementations may prepend nonces with a unique prefix. A unique Application instance identifier may serve such purpose.

12.2 `Announce` message content

An `Announce` message content type is identified by

```
trytes AnnounceTypeId = "MAM9CHANNEL9ANNOUNCE";
```

Corresponding `Content` is described by the following `PB3` type:

```
message Announce {
  absorb tryte msspk[81];
  absorb oneof {
```

```
    null empty = 0;
    tryte ntrupk[3072] = 1;
  }
  commit;
  squeeze external tryte hash[78];
  mssig(hash) sig;
}
```

The fields of `Announce` have the following meaning:

- `msspk` — Author's MSS public key;
- `empty` — signifies absence of Author's NTRU public key;
- `ntrupk` — Author's NTRU public key;
- `hash` — message hash-value to be signed;
- `sig` — signature of `hash` field produced with the MSS private key corresponding to `msspk`.

Note, absence Author's NTRU public key from the `Announce` message disables Subscribers from using subscription service provided by `Subscribe` and `Unsubscribe` messages.

Subscriber must establish trust relationship towards Author's NTRU public key via a secure channel. Such service is out of scope of this specification.

12.3 `ChangeKey` message content

A `ChangeKey` message content type is identified by

```
trytes ChangeKeyTypeId = "MAM9CHANNEL9CHANGEKEY";
```

Corresponding `Content` is described by the following `PB3` type:

```
message ChangeKey {
  join link msgid;
  absorb tryte msspk[81];
  commit;
  squeeze external tryte hash[78];
  mssig(hash) sig_with_msspk;
  mssig(hash) sig_with_linked_msspk;
}
```

The fields of `ChangeKey` have the following meaning:

- `msgid` — link to the message containing trusted MSS public key. This key is used to derive trust relationship to the `msspk` public key;
- `msspk` — s new MSS public key;
- `hash` — message hash value to be signed;
- `sig_with_msspk` — signature generated with the MSS private key corresponding to the public key contained in `msspk` field – proof of knowledge of private key;

- `sig_with_linked_msspk` — signature generated with the MSS private key corresponding to the trusted public key contained in the linked message.

12.4 `Keyload` message content

A `Keyload` message content type is identified by

```
trytes KeyloadTypeId = "MAM9CHANNEL9KEYLOAD";
```

Corresponding `Content` is described by the following `PB3` type:

```
message Keyload {
    join link msgid;
    absorb tryte nonce[27];
    skip repeated {
        fork;
        mask tryte id[27];
        absorb external tryte psk[81];
        commit;
        mask(key) tryte ekey[81];
    }
    skip repeated {
        fork;
        mask tryte id[27];
        ntrukem(key) tryte ekey[3072];
    }
    absorb external tryte key[81];
    commit;
}
```

The fields of `Keyload` have the following meaning:

- `nonce` — a nonce to be used with the key encapsulated in the keyload. A unique nonce allows for session keys to be reused;
- `id` — key (PSK or NTRU public key) identifier;
- `psk` — preshared key known to Author and to a Subscriber – legitimate recipient;
- `ekey` — masked session key; session key is either encrypted with `Spongos` or with `NTRU` layers;
- `key` — secret session key; Subscriber – legitimate recipient – decrypts it from `ekey` field.

12.5 `SignedPacket` message content

The `SignedPacket` message may be linked to any other message in the channel. It contains both plain and masked payloads. The message can only be signed and published by Author. Author must first publish corresponding public key certificate in either `Announce` or `ChangeKey` message. A `SignedPacket` message content type is identified by

```
trytes SignedPacketTypeId = "MAM9CHANNEL9SIGNEDPACKET";
```

Corresponding `Content` is described by the following `PB3` type:

```
message SignedPacket {
    join link msgid;
    absorb trytes public_payload;
    mask trytes masked_payload;
    commit;
    squeeze external tryte hash[78];
    mssig(hash) sig;
}
```

The fields of `SignedPacket` have the following meaning:

- `msgid` — link to the base message;
- `public_payload` — public part of payload;
- `masked_payload` — masked part of payload;
- `hash` — hash value to be signed;
- `sig` — message signature generated with one of Author's private keys.

12.6 `TaggedPacket` message content

The `TaggedPacket` message may be linked to any other message in the channel. It contains both plain and masked payloads. The message is authenticated with MAC and can be published by Author or by a Subscriber. A `TaggedPacket` message content type is identified by

```
trytes TaggedPacketTypeId = "MAM9CHANNEL9TAGGEDPACKET";
```

Corresponding `Content` is described by the following `PB3` type:

```
message TaggedPacket {
    join link msgid;
    absorb trytes public_payload;
    mask trytes masked_payload;
    commit;
    squeeze tryte mac[81];
}
```

The fields of `TaggedPacket` have the following meaning:

- `msgid` — link to the base message;
- `public_payload` — public part of payload;
- `masked_payload` — masked part of payload;
- `mac` — message authentication code.

12.7 Subscribe message content

A `Subscribe` message content type is identified by

```
trytes SubscribeTypeId = "MAM9CHANNEL9SUBSCRIBE";
```

Corresponding `Content` is described by the following `PB3` type:

```
message Subscribe {
    join link msgid[27];
    ntrukem(key) tryte unsubscribe_key[3072];
    commit;
    mask tryte ntrupk[3072];
    commit;
    squeeze tryte mac[27];
}
```

The fields of `Subscribe` have the following meaning:

- `msgid` — link to the `Announce` message containing Author's NTRU public key.
- `unsubscribe_key` — encapsulated secret key that serves as an encryption key and as a password to unsubscribe from the `Channel` Application instance;
- `ntrupk` — Subscriber's NTRU public key encrypted with `Spongos` layer;
- `mac` — authentication tag.

Author must maintain the resulting `spongos` state associated to the Subscriber's NTRU public key in order to identify a Subscriber willing to unsubscribe.

Note, `Subscribe` message doesn't include proof of possession of the NTRU private key. Such proof can be established in an interactive protocol by Author's request. Such protocol is out of scope.

12.8 Unsubscribe message content

A `Unsubscribe` message content type is identified by

```
trytes UnsubscribeTypeId = "MAM9CHANNEL9UNSUBSCRIBE";
```

Corresponding `Content` is described by the following `PB3` type:

```
message Unsubscribe {
    join link msgid[27];
    commit;
    squeeze tryte mac[27];
}
```

The fields of `Unsubscribe` have the following meaning:

- `msgid` — link to the `Subscribe` message published by Subscriber.
- `mac` — authentication tag proving knowledge of the `unsubscribe_key` from the `Subscribe` message.

12.9 Use cases

Message topology defines relationships between links to messages and `spongos` states. Linked messages may contain various proofs needed to unwrap and verify current message.

In general, `Channel` Application supports tree-like message topology as show in Figure 4.

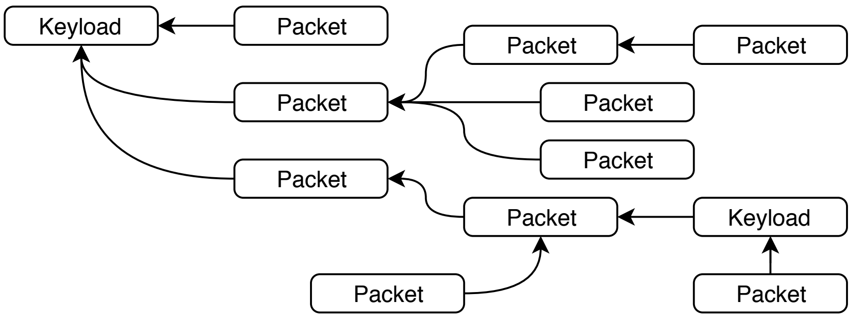


Figure 4: Tree-like message topology of `Channel` Application

However, maintaining such relationships may be too complex for devices with restricted resources.

List-like topology is shown in Figure 5 — the next `Packet` message is linked to the previous `Packet`. In this topology Subscriber must only maintain two `spongos` instances — current and linked — in order to wrap and unwrap packets. Such topology is also suitable for restricted Authors wishing to conserve MSS keys and only sign few packets. All previous tagged packets also become implicitly authenticated after publishing a signed packet. Random access to a packet, however, is linear in the number of previously linked packets.

Figure 5: List-like message topology of `Channel` Application

Set-like topology is shown in Figure 6 — the next `Packet` message is linked to the `Keyload` message. In this topology Subscriber must only maintain two `spongos` instances — current and associated to keyload — in order to wrap and unwrap packets. Such topology has advantage of having rapid random access to a packet.

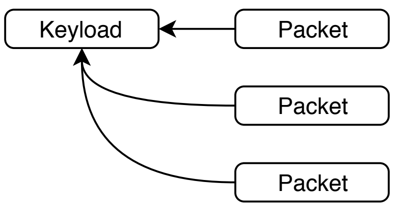


Figure 6: Set-like message topology of `Channel` Application

Public channel topology is shown in Figure 7 — no `Keyload` messages, all messages are public. In this topology Subscriber does not need to be subscribed. Both tagged and signed packets are published by the Author as Subscribers simply do not have key material to authenticate messages. Tagged messages are not authenticated, thus Author must end each chain of tagged messages with a signed one in order to authenticate the whole chain.

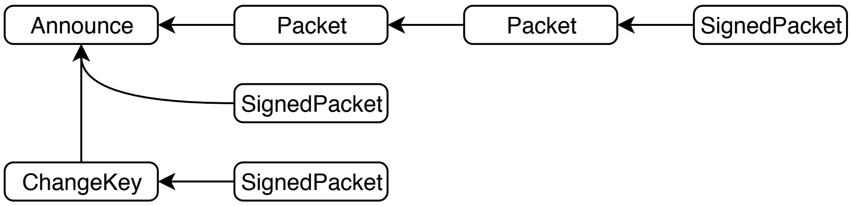


Figure 7: Public channel message topology

Short MT topology is shown in Figure 8 and trades off heavy resource requirements imposed by Merkle tree implementations with more frequent MSS key changes. In this topology a restricted Author maintains only one Merkle tree of small height. To be productive Merkle tree must contain at least three leaves as two leaves are used in `ChangeKey` messages: one key is used to prove possesion of private key when public key is published, and another one is used to sign a new public key. A restricted Subscriber needs only to maintain one Author's trusted MSS public key.

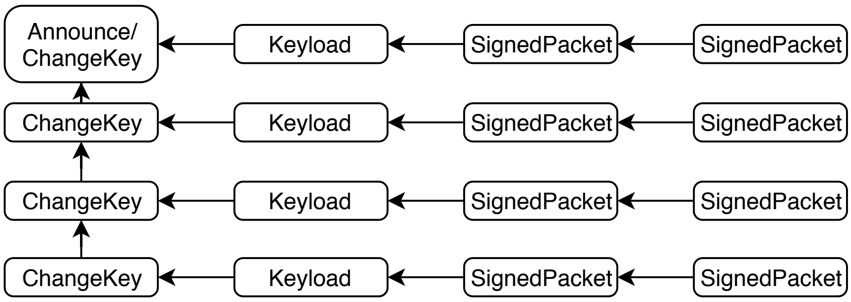


Figure 8: Short Merkel tree message topology

Distributed Author topology is shown in Figure 9 where Author wishes to employ several devices to publish signed packets. In this topology one device is selected to be master, it generates several new MSS private keys and publishes them in `ChangeKey` messages. It then distributes these MSS private keys between slave devices via a secure channel. These devices can now publish signed packets as well as establish their own hierarchy.

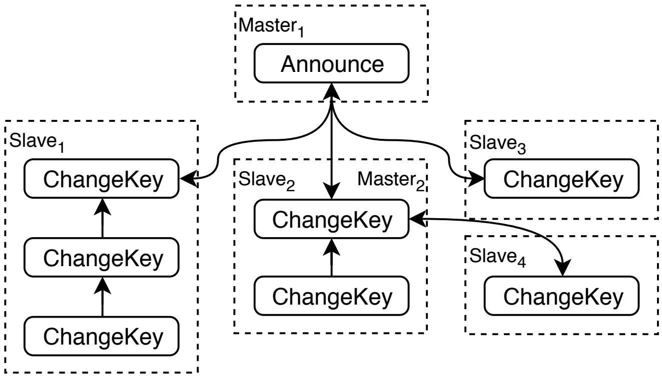


Figure 9: Distributed Author message topology