Workgroup:	Internet Engineering Task Force
Internet-Draft:	draft-gwerder-messagevortexmain-05
Published:	12 September 2020
Intended Status:	Experimental
Expires:	16 March 2021
Author:	M. Gwerder *FHNW*

MessageVortex Protocol

Abstract

The MessageVortex (referred to as Vortex) protocol achieves different degrees of anonymity, including sender, receiver, and third-party anonymity, by specifying messages embedded within existing transfer protocols, such as SMTP or XMPP, sent via peer nodes to one or more recipients.

The protocol outperforms others by decoupling the transport from the final transmitter and receiver. No trust is placed into any infrastructure except for that of the sending and receiving parties of the message. The creator of the routing block (Routing block builder;RBB) has full control over the message flow. Routing nodes gain no non-obvious knowledge about the messages even when collaborating. While third-party anonymity is always achieved, the protocol also allows for either sender or receiver anonymity.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Anonymisation is hard to achieve. Most previous attempts relied on either trust in a dedicated infrastructure or a specialized networking protocol.

Instead of defining a transport layer, Vortex piggybacks on other transport protocols. A blending layer embeds MessageVortex messages (VortexMessage) into ordinary messages of the respective transport protocol. This layer picks up the messages, passes them to a routing layer, which applies local operations to the messages, and resends the new message chunks to the next recipients.

A processing node learns as little as possible from the message or the network utilized. The operations have been designed to be sensible in any context. The 'onionized' structure of the protocol makes it impossible to follow the trace of a message without having control over the processing node.

MessageVortex is a protocol which allows sending and receiving messages by using a routing block instead of a destination address. With this approach, the sender has full control over all parameters of the message flow.

A message is split and reassembled during transmission. Chunks of the message may carry redundant information to avoid service interruptions during transit. Decoy and message traffic are not differentiable as the nature of the addRedundancy operation allows each generated portion to be either message or decoy. Therefore, any routing node is unable to distinguish between message and decoy traffic.

After processing, a potential receiver node knows if the message is destined for it (by creating a chunk with ID 0) or other nodes. Due to missing keys, no other node may perform this processing.

This RFC begins with general terminology (see Section 2) followed by an overview of the process (see Section 3). The subsequent sections describe the details of the protocol.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

1.2. Protocol Specification

Appendix A specifies all relevant parts of the protocol in ASN.1 (see [CCITT.X680.2002] and [CCITT.X208.1988]). The blocks are DER encoded, if not otherwise specified.

1.3. Number Specification

All numbers within this document are, if not suffixed, decimal numbers. Numbers suffixed with a small letter 'h' followed by two hexadecimal digits are octets written in hexadecimal. For example, a blank ASCII character (' ') is written as 20h and a capital 'K' in ASCII as 4Bh.

2. Entities Overview

The following entities used in this document are defined below.

2.1. Node

The term 'node' describes any computer system connected to other nodes, which support the MessageVortex Protocol. A 'node address' is typically an email address, an XMPP address or other transport protocol identity supporting the MessageVortex protocol. Any address SHOULD include a public part of an 'identity key' to allow messages to transmit safely. One or more addresses MAY belong to the same node.

2.1.1. Blocks

A 'block' represents an ASN.1 sequence in a transmitted message. We embed messages in the transport protocol, and these messages may be of any size.

2.1.2. NodeSpec

A nodeSpec block, as specified in Appendix A.6, expresses an addressable node in a unified format. The nodeSpec contains a reference to the routing protocol, the routing address within this protocol, and the keys required for addressing the node. This RFC specifies transport layers for XMPP and SMTP. Additional transport layers will require an extension to this RFC.

2.1.2.1. NodeSpec for SMTP nodes

An alternative address representation is defined that allows a standard email client to address a Vortex node. A node SHOULD support the smtpAlternateSpec (its specification is noted in ABNF as in [RFC5234]). For applications with QR code support, an implementation SHOULD use the smtpUrl representation.

```
                                        localPart        = <local part of address>
domain           = <domain part of address>
email           = localPart "@" domain
keySpec          = <BASE 6  4  encoded AsymmetricKey [DER encoded]>
smtpAlternateSpec = localPart ".." keySpec ".." domain "@localhost"
smtpUrl          = "vortexsmtp://" smtpAlternateSpec
```

This representation does not support quoted local part SMTP addresses.

2.1.2.2. NodeSpec for XMPP nodes

Typically, a node specification follows the ASN.1 block NodeSpec. For support of XMPP clients, an implementation SHOULD support the jidAlternateSpec (its specification is noted in ABNF as in [RFC5234]).

```
localPart       = <local part of address>
domain          = <domain part of address>
resourcePart    = <resource part of the address>
jid             = localPart "@" domain [ "/" resourcePart ]
keySpec         = <BASE64 encoded AsymmetricKey [DER encoded]>;
jidAlternateSpec  = localPart ".." keySpec ".."
                  domain "@localhost" [ "/" resourcePart ]
jidUrl          = "vortexxmpp://" jidAlternateSpec
```

2.2. Peer Partners

This document refers to two or more message sending or receiving entities as peer partners. One partner sends a message, and all others receive one or more messages. Peer partners are message specific, and each partner always connects directly to a node.

2.3. Encryption keys

Several keys are required for a Vortex message. For identities and ephemeral identities (see below), we use asymmetric keys, while symmetric keys are used for message encryption.

2.3.1. Identity Keys

Every participant of the network includes an asymmetric key, which SHOULD be either an EC key with a minimum length of 384 bits or an RSA key with a minimum length of 2048 bits.

The public key must be known by all parties writing to or through the node.

2.3.2. Peer Key

Peer keys are symmetrical keys transmitted with a Vortex message and are always known to the node sending the message, the node receiving the message, and the creator of the routing block.

A peer key is included in the Vortex message as well as the building instructions for subsequent Vortex messages (see RoutingCombo in Appendix A).

2.3.3. Sender Key

The sender key is a symmetrical key protecting the identity and routing block of a Vortex message. It is encrypted with the receiving peer key and prefixed to the identity block. This key further decouples the identity and processing information from the previous key.

A sender key is known to only one peer of a Vortex message and the creator of the routing block.

2.4. Vortex Message

The term 'Vortex message' represents a single transmission between two routing layers. A message adapted to the transport layer by the blending layer is called a 'blended Vortex message' (see Section 3).

A complete Vortex message contains the following items:

- The peer key, which is encrypted with the host key of the node and stored in a prefixBlock, protects the inner Vortex message (innerMessageBlock).
- The sender key, also encrypted with the host key of the node, protects the identity and routing block.
- The identity block, protected by the sender key, contains information about the ephemeral identity of the sender, replay protection information, header requests (optional), and a requirement reply (optional).
- The routing block, protected by the sender key, contains information on how subsequent messages are processed, assembled, and blended.
- The payload block, protected by the peer key, contains payload chunks for processing.

2.5. Message

A message is content to be transmitted from a single sender to a recipient. The sender uses a routing block either built itself or provided by the receiver to perform the transmission. While a message may be anonymous, there are different degrees of anonymity as described by the following.

- If the sender of a message is not known to anyone else except the sender, then this degree is referred to as 'sender anonymity.'
- If the receiver of a message is not known to anyone else except the receiver, then the degree is 'receiver anonymity.'
- If an attacker is unable to determine the content, original sender, and final receiver, then the degree is considered 'third-party anonymity.'
- If a sender or a receiver may be determined as one of a set of <k> entities, then it is referred to as k-anonymity[KAnon].

A message is always MIME encoded as specified in [RFC2045].

2.6. Key and MAC specifications and usage

MessageVortex uses a unique encoding for keys. This encoding is designed to be small and flexible while maintaining a specific base structure.

The following key structures are available:

- SymmetricKey
- AsymmetricKey

MAC does not require a complete structure containing specs and values, and only a MacAlgorithmSpec is available. The following sections outline the constraints for specifying parameters of these structures where a node MUST NOT specify any parameter more than once.

If a crypto mode is specified requiring an IV, then a node MUST provide the IV when specifying the key.

2.6.1. Asymmetric Keys

Nodes use asymmetric keys for identifying peer nodes (i.e., identities) and encrypting symmetric keys (for subsequent de-/encryption of the payload or blocks). All asymmetric keys MUST contain a key type specifying a strictly-normed key. Also, they MUST contain a public part of the key encoded as an X.509 container and a private key specified in PKCS#8 wherever possible.

RSA and EC keys MUST contain a keySize parameter. All asymmetric keys SHOULD contain a padding parameter, and a node SHOULD assume PKCS#1 if no padding is specified.

NTRU specification MUST provide the parameters "n", "p", and "q".

2.6.2. Symmetric Keys

Nodes use symmetric keys for encrypting payloads and control blocks. These symmetric keys MUST contain a key type specifying a key, which MUST be in an encoded form.

A node MUST provide a keySize parameter if the key (or, equivalently, the block) size is not standardized or encoded in the name. All symmetric key specifications MUST contain a mode and padding parameter. A node MAY list multiple padding or mode parameters in a ReplyCapability block to offer the recipient a free choice.

2.7. Transport Address

The term 'transport address' represents the token required to address the next immediate node on the transport layer. An email transport layer would have SMTP addresses, such as 'vortex@example.com,' as the transport address.

2.8. Identity

2.8.1. Peer Identity

The peer identity may contain the following information of a peer partner:

- A transport address (always) and the public key of this identity, given there is no recipient anonymity.
- A routing block, which may be used to contact the sender. If striving for recipient anonymity, then this block is required.
- The private key, which is only known by the owner of the identity.

2.8.2. Ephemeral Identity

Ephemeral identities are temporary identities created on a single node. These identities MUST NOT relate to another identity on any other node so that they allow bookkeeping for a node. Each ephemeral identity has a workspace assigned, and may also have the following items assigned.

- An asymmetric key pair to represent the identity.

- A validity time of the identity.

2.8.3. Official Identity

An official identity may have the following items assigned.

- Routing blocks used to reply to the node.
- A list of assigned ephemeral identities on all other nodes and their projected quotas.
- A list of known nodes with the respective node identity.

2.9. Workspace

Every official or ephemeral identity has a workspace, which consists of the following elements.

- Zero or more routing blocks to be processed.
- Slots for a payload block sequentially numbered. Every slot:
  - MUST contain a numerical ID identifying the slot.
  - MAY contain payload content.
  - If a block contains a payload, then it MUST contain a validity period.

2.10. Multi-use Reply Blocks

'Multi-use reply blocks' (MURB) are a special type routing block sent to a receiver of a message or request. A sender may use such a block one or several times to reply to the sender linked to the ephemeral identity, and it is possible to achieve sender anonymity using MURBs.

A vortex node MAY deny the use of MURBs by indicating a maxReplay equal to zero when sending a ReplyCapability block. An unobservable node SHOULD deny the use of MURBs.

2.11. Protocol Version

This Document describes the version 1 of the protocol. The message PrefixBlock contains an optional version indicator. If absent protocol version 1 should be assumed.

3. Layer Overview

The protocol is designed in four layers as shown in Figure 1.

```
                              +------------------------------------------------------------------+
| Vortex Node                                        |
| +------------------------------------------------------------------+ |
| |                  Accounting                      | |
| |___________________________________________________________| |
|                                                    |
| +------------------------------------------------------------------+ |
| |                  Routing                       | |
| |___________________________________________________________| |
|                                                    |
| +----------------------------+ +----------------------------------+ |
| |        Blending       | |         Blending         | |
| |__________________________| |______________________________| |
|______________________________________________________________|
 +----------------------------+ +----------------+ +------------------+
 |        Transport       | | Transport in | | Transport out |
 |__________________________| |_____________| |______________|
```

Figure 1: Layer overview

Every participating node MUST implement the layer's blending, routing, and accounting. There MUST be at least one incoming and one outgoing transport layer available to a node. All blending layers SHOULD connect to the respective transport layers for sending and receiving packets.

3.1. Transport Layer

The transport layer transfers the blended Vortex messages to the next vortex node and stores it until the next blending layer picks up the message.

The transport layer infrastructure SHOULD NOT be specific to anonymous communication and should contain significant portions of non-Vortex traffic.

3.2. Blending Layer

The blending layer embeds blended Vortex Message into the transport layer data stream and extracts the packets from the transport layer.

3.3. Routing Layer

The routing layer expands the information contained in MessageVortex packets, processes them, and passes generated packets to the respective blending layer.

3.4. Accounting Layer

The accounting layer tracks all ephemeral identities authorized to use a MessageVortex node and verifies the available quotas to an ephemeral identity.

4. Vortex Message

4.1. Overview

Figure 2 shows a Vortex message. The enclosed sections denote encrypted blocks, and the three or four-letter abbreviations denote the key required for decryption. The abbreviation k_h stands for the asymmetric host key, and sk_p is the symmetric peer key. The receiving node obtains this key by decrypting MPREFIX with its host key k_h. Then, sk_s is the symmetric sender key. When decrypting the MPREFIX block, the node obtains this key. The sender key protects the header and routing blocks by guaranteeing the node assembling the message does not know about upcoming identities, operations, and requests. The peer key protects the message, including its structure, from third-party observers.

```
                        +-+---+-+-+---+-+---+-+-+---+-+-+---+-+-------+-+
| |  | | | | C | | |  | | | R | |     | |
| |  | | | | P | | | H | | | O | |     | |
| | M | | | | R | | | E | | | U | |  P  | |
| | P | | | | E | | | A | | | T | |  A  | |
| | R | | | | F | | | D | | | I | |  Y  | |
| | E | | | | I | | | E | | | N | |  L  | |
| | F | | | | X | | | R | | | G | |  O  | |
| | I | | | +---+ | |___| | |___| |  A  | |
| | X | | |  k_h  | sk_s  | sk_s  |  D  | |
| |___| | |_______|_______|_______|_______| |
|  k_h  |           sk_p            |
|_______|___________________________________|
```

Figure 2: Vortex message overview

4.2. Message Prefix Block (MPREFIX)

The PrefixBlock contains a symmetrical key as defined in Appendix A.1 and is encrypted using the host key of the receiving peer host. The symmetric key utilized MUST be from the set advertised by a CapabilitiesReplyBlock (see Section 7.2.6). A node MAY choose any parameters omitted in the CapabilitiesReplyBlock freely unless stated otherwise in Section 7.2.6. A node SHOULD avoid sending unencrypted PrefixBlocks, and a prefix block MUST contain the same forward-secret as the other prefix as well as the routing and header blocks. A host MAY reply to a message with an unencrypted message block, but any reply to a message SHOULD be encrypted.

The sender MUST choose a key which may be encrypted with the host key in the respective PrefixBlock using the padding advertised by the CapabilitiesReplyBlock.

4.3. Inner Message Block

A node MUST always encrypt an InnerMessageBlock with the symmetric key of the PrefixBlock to hide the inner structure of the message. The InnerMessageBlock SHOULD always accommodate four or more payload chunks.

4.3.1. Control Prefix Block

Control prefix (CPREFIX) and MPREFIX blocks share the same structure and logic as well as containing the sender key sk_s. If an MPREFIX block is unencrypted, a node MAY omit the CPREFIX block. An omitted CPREFIX block results in unencrypted control blocks (e.g., the HeaderBlock and RoutingBlock).

4.3.2. Control Blocks

The control blocks of the HeaderBlock and a RoutingBlock contain the core information to process the payload.

4.3.2.1. Header Block

The header block (see HeaderBlock in Appendix A) contains the following information.

- It MUST contain the local ephemeral identity of the routing block builder.
- It MAY contain header requests.
- It MAY contain the solution to a PuzzleRequired block previously opposed in a header request.

The list of header requests MAY be one of the following.

- Empty.
- Contain a single identity create request (HeaderRequestIdentity).
- Contain a single increase quota request.

If a header block violates these rules, then a node MUST NOT reply to any header request. The payload and routing blocks SHOULD still be added to the workspace and processed if the message quota is not exceeded.

4.3.2.2. Routing Block

The routing block (see RoutingBlock in Appendix A) contains the following information.

- It MUST contain a serial number uniquely identifying the routing block of this user. The serial number MUST be unique during the lifetime of the routing block.
- It MUST contain the same forward secret as the two prefix blocks and the header block.
- It MAY contain assembly and processing instructions for subsequent messages.
- It MAY contain a reply block for messages assigned to the owner of the identity.

4.3.3. Payload Block

Each InnerMessageBlock with routing information SHOULD contain at least four PayloadChunks.

5. General notes

The MessageVortex protocol is a modular protocol that allows the use of different encryption algorithms. For its operation, a Vortex node SHOULD always support at least two distinct types of algorithms, paddings or modes such that they rely on two mathematical problems.

5.1. Supported Symmetric Ciphers

A node MUST support the following symmetric ciphers.

- AES128 (see [FIPS-AES] for AES implementation details).
- AES256.
- CAMELLIA128 (see [RFC3657] Chapter 3 for Camellia implementation details).
- CAMELLIA256.

A node SHOULD support any standardized key larger than the smallest key size.

A node MAY support Twofish ciphers (see [TWOFISH]).

5.2. Supported Asymmetric Ciphers

A node MUST support the following asymmetric ciphers.

- RSA with key sizes greater or equal to 2048 ([RFC8017]).
- ECC with named curves secp384r1, sect409k1 or secp521r1 (see [SEC1]).

5.3. Supported MACs

A node MUST support the following Message Authentication Codes (MAC).

- SHA3-256 (see [ISO-10118-3] for SHA implementation details).
- RipeMD160 (see [ISO-10118-3] for RIPEMD implementation details).

A node SHOULD support the following MACs.

- SHA3-512.
- RipeMD256.
- RipeMD512.

5.4. Supported Paddings

A node MUST support the following paddings specified in [RFC8017].

- PKCS1 (see [RFC8017]).
- PKCS7 (see [RFC5958]).

5.5. Supported Modes

A node MUST support the following modes.

- CBC (see [RFC1423]) such that the utilized IV must be of equal length as the key.
- EAX (see [EAX]).
- GCM (see [RFC5288]).
- NONE (only used in special cases, see Section 11).

A node SHOULD NOT use the following modes.

- NONE (except as stated when using the addRedundancy function).
- ECB.

A node SHOULD support the following modes.

- CTR ([RFC3686]).
- CCM ([RFC3610]).
- OCB ([RFC7253]).
- OFB ([MODES]).

6. Blending

Each node supports a fixed set of blending capabilities, which may be different for incoming and outgoing messages.

The following sections describe the blending mechanism. There are currently two blending layers specified with one for the Simple Mail Transfer Protocol (SMTP, see [RFC5321]) and the second for the Extensible Messaging and Presence Protocol (XMPP, see [RFC6120]). All nodes MUST at least support "encoding=plain:0,256".

6.1. Blending in Attachments

There are two types of blending supported when using attachments.

- Plain binary encoding with offset (PLAIN).
- Embedding with F5 in an image (F5).

A node MUST support PLAIN blending for reasons of interoperability whereas a node MAY support blending using F5.

A routing block builder (RBB) MUST take care about sizing restrictions of the transport layer when composing routing blocks

6.1.1. PLAIN embedding into attachments

A blending layer embeds a VortexMessage in a carrier file with an offset for PLAIN blending. For replacing a file start, a node MUST use the offset 0. The routing node MUST choose the payload file for the message, and SHOULD use a credible payload type (e.g., MIME type) with high entropy. Furthermore, it SHOULD prefix a valid header structure to avoid easy detection of the Vortex message. Finally, a routing node SHOULD use a valid footer, if any, to a payload file to improve blending.

The blended Vortex message is embedded in one or more message chunks, each starting with a chnk header. The chunk header consists two unsigned integers of variable length. The integer starts with the LSB, and if bit 7 is set, then there is another byte following. There cannot be more than four bytes where the last, fourth byte is always 8 bit. The three preceding bytes have a payload of seven bits each, which results in a maximum number of 2^29 bits. The first of the extracted numbers (modulo remaining document bytes starting from the first and including byte of the chunk header) reflect the number of bytes in the chunk after the chunk header. The second contains the number of bytes (again modulo remaing document bytes) to be skipped after the current chunk to reach the next chunk. There exists no "last chunk" indicator. And a gap or chunk may surpass the end of the file.

```
position: 00h 02h 04h 06h 08h ... 400h 402h 404h 406h 408h
 40Ah
value:  01 02 03 04 05 06 07 08 09 ... 01 05 0A 0B 0C 0D 0E 0Ff0
 03 12 13

Embedding: "(plain:1024)"

Result:  0A 13 (+494 omitted bytes; then skip 12 bytes to next chunk)
```

A node SHOULD offer at least one PLAIN blending method and MAY offer multiple offsets for incoming Vortex messages.

A plain blending is specified as the following.

```
                        plainEncoding = "("plain:" <numberOfBytesOfOffset>
      [ "," <numberOfBytesOfOffset> ] *  ")"
```

6.1.2. F5 embedding into attachments

For F5, a blending layer embeds a Vortex message into a jpeg file according to [F5]. The password for blending may be public, and a routing node MAY advertise multiple passwords. The use of F5 adds approximately tenfold transfer volume to the message. A routing block building node SHOULD only use F5 blending where appropriate.

A blending in F5 is specified as the following.

```
                                      f 5 Encoding = "(F 5 :" <passwordString> [ ","
<PasswordString> ] *  ")"
```

Commas and backslashes in passwords MUST be escaped with a backslash whereas closing brackets are treated as normal password characters unless they are the final character of the encoding specification string.

6.2. Blending into an SMTP layer

Email messages with content MUST be encoded with Multipurpose Internet Mail Extensions (MIME) as specified in [RFC2045]. All nodes MUST support BASE64 encoding and MUST test all sections of a MIME message for the presence of a VortexMessage.

A vortex message is present if a block containing the peer key at the known offset of any MIME part decodes correctly.

A node SHOULD support SMTP blending for sending and receiving. For sending SMTP, the specification in [RFC5321] must be used. TLS layers MUST always be applied when obtaining messages using POP3 (as specified in [RFC1939] and [RFC2595]) or IMAP (as specified in [RFC3501]). Any SMTP connection MUST employ a TLS encryption when passing credentials.

6.3. Blending into an XMPP layer

For interoperability, an implementation SHOULD provide XMPP blending.

Blending into XMPP traffic is performed using the [XEP-0231] extension of the XMPP protocol.

PLAIN and F5 blending are acceptable for this transport layer.

7. Routing

7.1. Vortex Message Processing

7.1.1. Processing of incoming Vortex Messages

An incoming message is considered initially unauthenticated. A node should consider a VortexMessage as authenticated as soon as the ephemeral identity is known and is not temporary.

For an unauthenticated message, the following rules apply.

- A node MUST ignore all Routing blocks.
- A node MUST ignore all Payload blocks.
- A node SHOULD accept identity creation requests in unauthenticated messages.
- A node MUST ignore all other header requests except identity creation requests.
- A node MUST ignore all identity creation requests belonging to an existing identity.

A message is considered authenticated as soon as the identity used in the header block is known and not temporary. A node MUST NOT treat a message as authenticated if the specified maximum number of replays is reached. For authenticated messages, the following rules apply.

- A node MUST ignore identity creation requests.
- A node MUST replace the current reply block with the reply block provided in the routing block (if any). The node MUST keep the reply block if none is provided.
- A node SHOULD process all header requests.
- A node SHOULD add all routing blocks to the workspace.
- A node SHOULD add all payload blocks to the workspace.

A routing node MUST decrement the message quota by one if a received message is authenticated, valid, and contains at least one payload block. If a message is identified as duplicate according to the reply protection, then a node MUST NOT decrement the message quota.

The message processing works according pseudo-code shown below.

```
                                        function incomming_message(VortexMessage
blendedMessage) {
  try{
    msg = unblend( blendedMessage );
    if( not msg ) {
      // Abort processing
      throw exception( "no embedded message found" )
    } else {
      hdr = get_header( msg )
      if( not known_identity( hdr.identity ) {
        if( get_requests( hdr ) contains HeaderRequestIdentity ) {
          create_new_identity( hdr ).set_temporary( true )
          send_message( create_requirement( hdr )  )
        } else {
          // Abort processing
          throw exception( "identity unknown" )
        }
      } else {
        if( is_duplicate_or_replayed( msg ) ) {
          // Abort processing
          throw exception "duplicate or replayed message" )
        } else {
          if( get_accounting( hdr.identity ).is_temporary() ) {
            if( not verify_requirement( hdr.identity, msg ) ) {
              get_accounting( hdr.identity ).set_temporary( false )
            }
          }
          if( get_accounting( hdr ).is_temporary() ) {
            throw exception( "no processing on temporary identity" )
          }

          // Message authenticated
          get_accounting( hdr.identity ).register_for_replay_protection( msg )
          if( not verify_mtching_forward_secrets( msg ) ) {
            throw exception( "forward secret missmatch" )
          }
          if( contains_payload( msg ) ) {
            if( get_accounting( hdr.identity ).decrement_message_quota() ) {
              while index,nextPayloadBlock = get_next_payload_block( msg ) {
                add_workspace( header.identity, index, nextPayloadBlock )
              }
              while nextRoutingBlock = get_next_routing_block( msg ) {
                add_workspace( hdr.identity, add_routing( nextRoutingBlock ) )
              }
              process_reserved_mapping_space( msg )
              while nextRequirement = get_next_requirement( hdr ) {
                add_workspace( hdr.identity, nextRequirement )
              }
            } else {
              throw exception( "Message quota exceeded" )
            }
          }
```

```
      }
    }
  } catch( exception e ) {
    // Message processing failed
    throw e;
  }
}
```

7.1.2. Processing of Routing Blocks in the Workspace

A routing workspace consists of the following items.

- The identity linked to, which determines the lifetime of the workspace.
- The linked routing combos (RoutingCombo).
- A payload chunk space with the following multiple subspaces available:
  - ID 0 represents a message to be embedded (when reading) or a message to be extracted to the user (when written).
  - ID 1 to ID maxPayloadBlocks represent the payload chunk slots in the target message.
  - All blocks between ID maxPayloadBlocks + 1 to ID 32766 belong to a temporary routing block-specific space.
  - ID 32767 MUST be used to signal a solicited reply block.
  - All blocks between ID 32768 to ID 65535 belong to a shared space available to all operations of the identity.

The accounting layer typically triggers processing and represents either a cleanup action or a routing event. A cleanup event deletes the following information from all workspaces.

- All processed routing combos.
- All routing combos with expired usagePeriod.
- All payload chunks exceeding the maxProcess time.
- All expired objects.
- All expired puzzles.
- All expired identities.
- All expired replay protections.

Note that maxProcessTime reflects the number of seconds since the arrival of the last octet of the message at the transport layer facility. A node SHOULD NOT take additional processing time (e.g., for anti-UBE or anti-virus) into account.

The accounting layer triggers routing events occurring at least the minProcessTime after the last octet of the message arrived at the routing layer. A node SHOULD choose the latest possible moment at which the peer node receives the last octet of the assembled message before the maxProcessTime is reached. The calculation of this last point in time where a message may be set

SHOULD always assume that the target node is working. A sending node SHOULD choose the time within these bounds randomly. An accounting layer MAY trigger multiple routing combos in bulk to further obfuscate the identity of a single transport message.

First, the processing node escapes the payload chunk at ID 0 if needed (e.g., a non-special block is starting with a backslash). Next, it executes all processing instructions of the routing combo in the specified sequence. If an instruction fails, then the block at the target ID of the operation remains unchanged. The routing layer proceeds with the subsequent processing instructions by ignoring the error. For a detailed description of the operations, see Section 7.4. If a node succeeds in building at least one payload chunk, then a VortexMessage is composed and passed to the blending layer.

7.1.3. Processing of Outgoing Vortex Messages

The blending layer MUST compose a transport layer message according to the specification provided in the routing combo. It SHOULD choose any decoy message or steganographic carrier in such a way that the dead parrot syndrome, as specified in [DeadParrot], is avoided.

7.2. Header Requests

Header requests are control requests for the anonymization system. Messages with requests or replies only MUST NOT affect any quota.

7.2.1. Request New Ephemeral Identity

Requesting a new ephemeral identity is performed by sending a message containing a header block with the new identity and an identity creation request (HeaderRequestIdentity) to a node. The node MAY send an error block (see Section 7.3.1) if it rejects the request.

If a node accepts an identity creation request, then it MUST send a reply. A node accepting a request without a requirement MUST send back a special block containing "no error". A node accepting a request under the precondition of a requirement to be fulfilled MUST send a special block containing a requirement block.

A node SHOULD NOT reply to any clear-text requests if the node does not want to disclose its identity as a Vortex node officially. A node MUST reply with an error block if a valid identity is used for the request.

7.2.2. Request Message Quota

Any valid ephemeral identity may request an increase of the current message quota to a specific value at any time. The request MUST include a reply block in the header and may contain other parts. If a requested value is lower than the current quota, then the node SHOULD NOT refuse the quota request and SHOULD send a "no error" status.

A node SHOULD reply to a HeaderRequestIncreaseMessageQuota request (see Appendix A) of a valid ephemeral identity. The reply MUST include a requirement, an error message or a "no error" status message.

7.2.3. Request Increase of Message Quota

A node may request to increase the current message quota by sending a HeaderRequestIncreaseMessageQuota request to the routing node. The value specified within the node is the new quota. HeaderRequestIncreaseMessageQuota requests MUST include a reply block, and a node SHOULD NOT use a previously sent MURB to reply.

If the requested quota is higher than the current quota, then the node SHOULD send a "no error" reply. If the requested quota is not accepted, then the node SHOULD send a requestedQuotaOutOfBand reply.

A node accepting the request MUST send a RequirementBlock or a "no error block."

7.2.4. Request Transfer Quota

Any valid ephemeral identity may request to increase the current transfer quota to a specific value at any time. The request MUST include a reply block in the header and may contain other parts. If a requested value is lower than the current quota, then the node SHOULD NOT refuse the quota request and SHOULD send a "no error" status.

A node SHOULD reply to a HeaderRequestIncreaseTransferQuota request (see Appendix A) of a valid ephemeral identity. The reply MUST include a requirement, an error message or a "no error" status message.

7.2.5. Query Quota

Any valid ephemeral identity may request the current message and transfer quota. The request MUST include a reply block in the header and may contain other parts.

A node MUST reply to a HeaderRequestQueryQuota request (see Appendix A), which MUST include the current message quota and the current message transfer quota. The reply to this request MUST NOT include a requirement.

7.2.6. Request Capabilities

Any node MAY request the capabilities of another node, which include all information necessary to create a parseable VortexMessage. Any node SHOULD reply to any encrypted HeaderRequestCapability.

A node SHOULD NOT reply to clear-text requests if the node does not want to disclose its identity as a Vortex node officially. A node MUST reply if a valid identity is used for the request, and it MAY reply to unknown identities.

7.2.7. Request Nodes

A node may ask another node for a list of routing node addresses and keys, which may be used to bootstrap a new node and add routing nodes to increase the anonymization of a node. The receiving node of such a request SHOULD reply with a requirement (e.g., RequirementPuzzleRequired).

A node MAY reply to a HeaderRequest request (see Appendix A) of a valid ephemeral identity, and the reply MUST include a requirement, an error message or a "no error" status message. A node MUST NOT reply to an unknown identity, and SHOULD always reply with the same result set to the same identity.

7.2.8. Request Identity Replace

This request type allows a receiving node to replace an existing identity with the identity provided in the message, and is required if an adversary manages to deny the usage of a node (e.g., by deleting the corresponding transport account). Any sending node may recover from such an attack by sending a valid authenticated message to another identity to provide the new transport and key details.

A node SHOULD reply to such a request from a valid known identity, and the reply MUST include an error message or a "no error" status message.

7.2.9. Request Upgrade

This request type allows a node to request a new version of the software in an anonymous, unliked manor. The identifier MUST identify the software product uniquely. The version MUST reflect the version tag of the currently installed version or a similarly usable tag.

7.3. Special Blocks

Special blocks are payload messages that reflect messages from one node to another and are not visible to the user. A special block starts with the character sequence '\special' (or 5Ch 73h 70h 65h 63h 69h 61h 6Ch) followed by a DER encoded special block (SpecialBlock). Any non-special message decoding to ID 0 in a workspace starting with this character sequence MUST escape all backslashes within the payload chunk with an additional backslash.

7.3.1. Error Block

An error block may be sent as a reply contained in the payload section. The error block is embedded in a special block and sent with any provided reply block. Error messages SHOULD contain the serial number of the offending header block and MAY contain human-readable text providing additional messages about the error.

7.3.2. Requirement Block

If a node is receiving a requirement block, then it MUST assume that the request block is accepted, is not yet processed, and is to be processed if it meets the contained requirement. A node MUST process a request as soon as the requirement is fulfilled, and MUST resend the request as soon as it meets the requirement.

A node MAY reject a request, accept a request without a requirement, accept a request upon payment (RequirementPaymentRequired), or accept a request upon solving a proof of work puzzle (RequirementPuzzleRequired).

7.3.2.1. Puzzle Requirement

If a node requests a puzzle, then it MUST send a RequirementPuzzleRequired block. The puzzle requirement is solved if the node receiving the puzzle is replying with a header block that contains the puzzle block, and the hash of the encoded block begins with the bit sequence mentioned in the puzzle within the period specified in the field 'valid.'

A node solving a puzzle requires sending a VortexMessage to the requesting node, which MUST contain a header block that includes the puzzle block and MUST have a MAC fingerprint starting with the bit sequence as specified in the challenge. The receiving node calculates the MAC from the unencrypted DER encoded HeaderBlock with the algorithm specified by the node. The sending node may achieve the requirement by adding a proofOfWork field to the HeaderBlock containing any content fulfilling the criteria. The sending node SHOULD keep the proofOfWork field as short as possible.

7.3.2.2. Payment Requirement

If a node requests a payment, then it MUST send a RequirementPaymentRequired block. As soon as the requested fee is paid and confirmed, the requesting node MUST send a "no error" status message. The usage period 'valid' describes the period during which the payment may be carried out. A node MUST accept the payment if occurring within the 'valid' period but confirmed later. A node SHOULD return all unsolicited payments to the sending address.

7.3.2.3. Upgrade

If a node requests an upgrade a ReplyUpgrade block MAY be sent. The block must contain the identifier and version of the most recent software version. The blob MAY contain the software if there is a newer one available.

7.4. Routing Operations

Routing operations are contained in a routing block and processed upon arrival of a message or when compiling a new message. All operations are reversible, and no operation is available for generating decoy traffic, which may be used through encryption of an unpadded block or the addRedundancy operation.

All payload chunk blocks inherit the validity time from the message routing combos as arrival time + max(maxProcessTime).

When applying an operation to a source block, the resulting target block inherits the expiration of the source block. When multiple expiration times exist, the one furthest in the future is applied to the target block. If the operation fails, then the target expiration remains unchanged.

7.4.1. Mapping Operation

The straightforward mapping operation is used in inOperations of a routing block to map the routing block's specific blocks to a permanent workspace.

7.4.2. Split and Merge Operations

The split and merge operations allow splitting and recombining message chunks. A node MUST adhere to the following constraints.

- The operation must be applied at an absolute (measuring in bytes) or relative (measured as a float value in the range 0>value>100) position.
- All calculations must be performed according to IEEE 754 [IEEE754] and in 64-bit precision.
- If a relative value is a non-integer result, then a floor operation (i.e., cutting off all non-integer parts) determines the number of bytes.
- If an absolute value is negative, then the size represents the number of bytes counted from the end of the message chunk.
- If an absolute value is greater than the number of bytes in a block, then all bytes are mapped to the respective target block, and the other target block becomes a zero byte-sized block.

An operation MUST fail if relative values are equal to, or less than, zero. An operation MUST fail if a relative value is equal to, or greater than, 100. All floating-point operations must be performed according to [IEEE754] and in 64-bit precision.

7.4.3. Encrypt and Decrypt Operations

Encryption and decryption are executed according to the standards mentioned above. An encryption operation encrypts a block symmetrically and places the result in the target block. The parameters MUST contain IV, padding, and cipher modes. An encryption operation without a valid parameter set MUST fail.

7.4.4. Add and Remove Redundancy Operations

The addRedundancy and removeRedundancy operations are core to the protocol. They may be used to split messages and distribute message content across multiple routing nodes. The operation is separated into three steps.

1. Pad the input block to a multiple of the key block size in the resulting output blocks.
2. Apply a Vandermonde matrix with the given sizes.
3. Encrypt each resulting block with a separate key.

The following sections describe the order of the operations within an addRedundancy operation. For a removeRedundancy operation, invert the functions and order. If the removeRedundancy has more than the required blocks to recover the information, then it should take only the required number beginning from the smallest. If a seed and PRNG are provided, then the removeRedundancy operation MAY test any combination until recovery is successful.

7.4.4.1. Padding Operation

Padding is done in multiple steps. First, we calculate the padding value p. We then concatenate the padding value p as 32 bit little-endian unint with the message and fill the remaining bytes required with the seeded PRNG.

A processing node calculates the final length of all payload blocks, including redundancy. This is done by in three steps followed by the calculation of the padding value p.

1. i=len(<input block>) [calculate the size of the input block]
2. e=lcm(<Blocksize of output encyrption in # bytes>,<# of output blocks>) [Calculate Minimum size of the output block]
3. l=roof((i+4+C2)/e)*e [Calculate the final length of the padded stream suitable for the subsequent operations. C2 is a constant which is either provided by the RBB or 0 if not specified.]
4. p=i+(C1*l(mod (roof((2^32-1-i)/l)*))) [Calculate padding value p. C1 is a positive integer constant and MUST be provided by the RBB to maintain diagnosability.]

The remainder of the input block, up to length L, is padded with random data. A routing block builder should specify the value of the $randomInteger$. If not specified the routing node may choose a random positive integer value. A routing block builder SHOULD specify a PRNG and a seed used for this padding. If GF(16) is applied, then all numbers are treated as little-endian representations. Only GF(8) and GF(16) are allowed fields.

The length of 0 is a valid length

This padding guarantees that each resulting block matches the block size of the subsequent encryption operation and does not require further padding.

For padding removal, the padding p at the start is first removed as a little-endian integer. Second, the length of the output block is calculated by applying <output block size in bytes>=p (mod <input block size in bytes>-4)

7.4.4.2. Apply Matrix

Next, the input block is organized in a data matrix D of dimensions (inrows, incols) where incols=(<number of data blocks>-<number of redundancy blocks>) and inrows=L/(<number of data blocks>-<number of redundancy blocks>). The input block data is first distributed in this matrix across, and then down.

Next, the data matrix D is multiplied by a Vandermonde matrix V with its number of rows equal to the incols calculated and columns equal to the <number of data blocks>. The content of the matrix is formed by v(i,j)=pow(i,j), where i reflects the row number starting at 0, and j reflects the column number starting at 0. The calculations described must be carried out in the GF noted in the respective operation to be successful. The completed operation results in matrix A.

7.4.4.3. Encrypt Target Block

Each row vector of A is a new data block encrypted with the corresponding encryption key noted in the keys of the addRedundancyOperation. If there are not enough keys available, then the keys used for encryption are reused from the beginning after the final key is used. A routing block builder SHOULD provide enough keys so that all target blocks may be encrypted with a unique key. All encryptions SHOULD NOT use padding.

7.5. Processing of Vortex Messages

The accounting layer triggers processing according to the information contained in a routing block in the workspace. All operations MUST be executed in the sequence provided in the routing block, and any failing operation must leave the result block unmodified.

All workspace blocks resulting in IDs of 1 to maxPayloadBlock are then added to the message and passed to the blending layer with appropriate instructions.

8. Accounting

8.1. Accounting Operations

The accounting layer has two types of operations.

- Time-based (e.g., cleanup jobs and initiation of routing).
- Routing triggered (e.g., updating quotas, authorizing operations, and pickup of incoming messages).

Implementations MUST provide sufficient locking mechanisms to guarantee the integrity of accounting information and the workspace at any time.

8.1.1. Time-Based Garbage Collection

The accounting layer SHOULD keep a list of expiration times. As soon as an entry (e.g., payload block or identity) expires, the respective structure should be removed from the workspace. An implementation MAY choose to remove expired items periodically or when encountering them during normal operation.

8.1.2. Time-Based Routing Initiation

The accounting layer MAY keep a list of when a routing block is activated. For improved privacy, the accounting layer should use a slotted model where, whenever possible, multiple routing blocks are handled in the same period, and the requests to the blending layers are mixed between the transactions.

8.1.3. Routing Based Quota Updates

A node MUST update quotas on the respective operations. For example, a node MUST decrease the message quota before processing routing blocks in the workspace and after the processing of header requests.

8.1.4. Routing Based Authorization

The transfer quota MUST be checked and decreased by the number of data bytes in the payload chunks after an outgoing message is processed and fully assembled. The message quota MUST be decreased by one on each routing block triggering the assembly of an outgoing message.

8.1.5. Ephemeral Identity Creation

Any packet may request the creation of an ephemeral identity. A node SHOULD NOT accept such a request without a costly requirement since the request includes a lifetime of the ephemeral identity. The costs for creating the ephemeral identity SHOULD increase if a longer lifetime is requested.

9. Acknowledgments

Thanks go to my family who supported me with patience and countless hours as well as to Mark Zeman for his feedback challenging my thoughts and peace.

10. IANA Considerations

This memo includes no request to IANA.

Additional encryption algorithms, paddings, modes, blending layers or puzzles MUST be added by writing an extension to this or a subsequent RFC. For testing purposes, IDs above 1,000,000 should be used.

11. Security Considerations

The MessageVortex protocol should be understood as a toolset instead of a fixed product. Depending on the usage of the toolset, anonymity and security are affected. For a detailed analysis, see [MVAnalysis].

The primary goals for security within this protocol rely on the following focus areas.

- Confidentiality
- Integrity
- Availability
- Anonymity
  - Third-party anonymity
  - Sender anonymity
  - Receiver anonymity

These aspects are affected by the usage of the protocol, and the following sections provide additional information on how they impact the primary goals.

The Vortex protocol does not rely on any encryption of the transport layer since Vortex messages are already encrypted. Also, confidentiality is not affected by the protection mechanisms of the transport layer.

If a transport layer supports encryption, then a Vortex node SHOULD use it to improve the privacy of the message.

Anonymity is affected by the inner workings of the blending layer in many ways. A Vortex message cannot be read by anyone except the peer nodes and routing block builder. The presence of a Vortex node message may be detected through the typical high entropy of an encrypted file, broken structures of a carrier file, a meaningless content of a carrier file or the contextless communication of the transport layer with its peer partner. A blending layer SHOULD minimize the possibility of simply detection by minimizing these effects.

A blending layer SHOULD use carrier files with high compression or encryption. Carrier files SHOULD NOT have inner structures such that the payload is comparable to valid content. To achieve undetectability by a human reviewer, a routing block builder should use F5 instead of PLAIN blending. This approach, however, increases the protocol overhead by approximately tenfold.

The two layers of 'routing' and 'accounting' have the deepest insight into a Vortex message's inner working. Each knows the immediate peer sender and the peer recipients of all payload chunks. As decoy traffic is generated by combining chunks and applying redundancy calculations, a node can never know if a malfunction (e.g., during a recovery calculation) was intended. Therefore, a node is unable to distinguish a failed transaction from a terminated transaction as well as content from decoy traffic.

A routing block builder SHOULD follow the following rules not to compromise a Vortex message's anonymity.

- All operations applied SHOULD be credibly involved in a message transfer.
- A sufficient subset of the result of an addRedundancy operation should always be sent to peers to allow recovery of the data built.
- The anonymity set of a message should be sufficiently large to avoid legal prosecution of all jurisdictional entities involved, even if a certain amount of the anonymity set cooperates with an adversary.
- Encryption and decryption SHOULD follow normal usage whenever possible by avoiding the encryption of a block on a node with one key and decrypting it with a different key on the same or adjacent node.
- Traffic peaks SHOULD be uniformly distributed within the entire anonymity set.
- A routing block SHOULD be used for a limited number of messages. If used as a message block for the node, then it should be used only once. A block builder SHOULD use the HeaderRequestReplaceIdentity block to update the reply to routing blocks regularly. Implementers should always remember that the same routing block is identifiable by its structure.

An active adversary cannot use blocks from other routing block builders. While the adversary may falsify the result by injecting an incorrect message chunk or not sending a message, such message disruptions may be detected by intentionally routing information to the routing block builder (RBB) node. If the Vortex message does not carry the information expected, then the node may safely assume that one of the involved nodes is misbehaving. A block building node MAY calculate reputation for involved nodes over time and MAY build redundancy paths into a routing block to withstand such malicious nodes.

Receiver anonymity is at risk if the handling of the message header and content is not done with care. An attacker might send a bugged message (e.g., with a DKIM or DMARC header) to deanonymize a recipient. Careful attention is required when handling anything other than local references when processing, verifying, or rendering a message.

12. References

12.1. Normative References

[CCITT.X208.1988] International Telephone and Telegraph Consultative Committee, "Specification of Abstract Syntax Notation One (ASN.1)", CCITT Recommendation X.208, November 1998.

[CCITT.X680.2002] International Telephone and Telegraph Consultative Committee, "Abstract Syntax Notation One (ASN.1): Specification of basic notation", November 2002.

[EAX] Bellare, M., Rogaway, P., and D. Wagner, "The EAX mode of operation", 2011.

[F5] Westfeld, A., "F5 - A Steganographic Algorithm - High Capacity Despite Better Steganalysis", 24 October 2001.

[FIPS-AES] Federal Information Processing Standard (FIPS), "Specification for the ADVANCED ENCRYPTION STANDARD (AES)", November 2011.

[IEEE754] IEEE, "754-2008 - IEEE Standard for Floating-Point Arithmetic", 29 August 2008.

[ISO-10118-3] International Organization for Standardization, "ISO/IEC 10118-3:2004 -- Information technology -- Security techniques -- Hash-functions -- Part 3: Dedicated hash-functions", March 2004.

[MODES] National Institute for Standards and Technology (NIST), "Recommendation for Block Cipher Modes of Operation: Methods and Techniques", December 2001.

[RFC1423] Balenson, D., "Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers", RFC 1423, DOI 10.17487/RFC1423, February 1993, <https://www.rfc-editor.org/info/rfc1423>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3610] Whiting, D., Housley, R., and N. Ferguson, "Counter with CBC-MAC (CCM)", RFC 3610, DOI 10.17487/RFC3610, September 2003, <https://www.rfc-editor.org/info/rfc3610>.

[RFC3657] Moriai, S. and A. Kato, "Use of the Camellia Encryption Algorithm in Cryptographic Message Syntax (CMS)", RFC 3657, DOI 10.17487/RFC3657, January 2004, <https://www.rfc-editor.org/info/rfc3657>.

[RFC3686] Housley, R., "Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP)", RFC 3686, DOI 10.17487/RFC3686, January 2004, <https://www.rfc-editor.org/info/rfc3686>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC5288] Salowey, J., Choudhury, A., and D. McGrew, "AES Galois Counter Mode (GCM) Cipher Suites for TLS", RFC 5288, DOI 10.17487/RFC5288, August 2008, <https://www.rfc-editor.org/info/rfc5288>.

[RFC5958] Turner, S., "Asymmetric Key Packages", RFC 5958, DOI 10.17487/RFC5958, August 2010, <https://www.rfc-editor.org/info/rfc5958>.

[RFC7253] Krovetz, T. and P. Rogaway, "The OCB Authenticated-Encryption Algorithm", RFC 7253, DOI 10.17487/RFC7253, May 2014, <https://www.rfc-editor.org/info/rfc7253>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/info/rfc8017>.

[SEC1] Certicom Research, "SEC 1: Elliptic Curve Cryptography", 21 May 2009.

[TWOFISH] Schneier, B., "The Twofish Encryptions Algorithm: A 128-Bit Block Cipher, 1st Edition", March 1999.

[XEP-0231] Peter, S.A. and P. Simerda, "XEP-0231: Bits of Binary", 3 September 2008, <https://xmpp.org/extensions/xep-0231.html>.

12.2. Informative References

[DeadParrot] Houmansadr, A., Burbaker, C., and V. Shmatikov, "The Parrot is Dead: Observing Unobservable Network Communications", 2013, <https://people.cs.umass.edu/~amir/papers/parrot.pdf>.

[KAnon] Ahn, L., Bortz, A., and N.J. Hopper, "k-Anonymous Message Transmission", 2003.

[MVAnalysis] Gwerder, M., "MessageVortex", 2018, <https://messagevortex.net/devel/messageVortex.pdf>.

[RFC1939] Myers, J. and M. Rose, "Post Office Protocol - Version 3", STD 53, RFC 1939, DOI 10.17487/RFC1939, May 1996, <https://www.rfc-editor.org/info/rfc1939>.

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996, <https://www.rfc-editor.org/info/rfc2045>.

[RFC2595] Newman, C., "Using TLS with IMAP, POP3 and ACAP", RFC 2595, DOI 10.17487/RFC2595, June 1999, <https://www.rfc-editor.org/info/rfc2595>.

[RFC3501] Crispin, M., "INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1", RFC 3501, DOI 10.17487/RFC3501, March 2003, <https://www.rfc-editor.org/info/rfc3501>.

[RFC5321] Klensin, J., "Simple Mail Transfer Protocol", RFC 5321, DOI 10.17487/RFC5321, October 2008, <https://www.rfc-editor.org/info/rfc5321>.

[RFC6120] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 6120, DOI 10.17487/RFC6120, March 2011, <https://www.rfc-editor.org/info/rfc6120>.

Appendix A. The ASN.1 schema for Vortex messages

The following sections contain the ASN.1 modules specifying the MessageVortex Protocol.

A.1. The Main MessageVortex Blocks

```
MessageVortex-Schema DEFINITIONS EXPLICIT TAGS ::=
BEGIN
  EXPORTS PrefixBlock, InnerMessageBlock, RoutingBlock,
          maxWorkspaceID;
  IMPORTS SymmetricKey, AsymmetricKey, MacAlgorithmSpec, CipherSpec
              FROM MessageVortex-Ciphers
          HeaderRequest
              FROM MessageVortex-Requests
          PayloadOperation, MapBlockOperation
              FROM MessageVortex-Operations

          UsagePeriod, BlendingSpec
              FROM MessageVortex-Helpers;

  --
*******************************************
***********************
  -- Constant definitions
  --
*******************************************
***********************
  -- maximum serial number
  maxSerial               INTEGER ::= 4294967295
  -- maximum number of administrative requests
  maxNumOfRequests        INTEGER ::= 8
  -- maximum number of seconds which the message might be delayed
  -- in the local queue (starting from startOffset)
  maxDurationOfProcessing INTEGER ::= 86400
  -- maximum id of an operation
  minWorkspaceID          INTEGER ::= 32768
  -- maximum number of routing blocks in a message
  maxRoutingBlks          INTEGER ::= 127
  -- maximum number a block may be replayed
  maxNumOfReplays         INTEGER ::= 127
  -- maximum number of payload chunks in a message
  maxPayloadBlks          INTEGER ::= 127
  -- maximum number of seconds a proof of non revocation may be old
  maxTimeCachedProof      INTEGER ::= 86400
  -- The maximum ID of the workspace
  maxWorkspaceId          INTEGER ::= 65535
  -- The maximum number of assembly instructions per combo
  maxAssemblyInstr        INTEGER ::= 255


  --
*******************************************
***********************
  -- Types
  --
*******************************************
***********************
  PuzzleIdentifier ::= OCTET STRING ( SIZE(0..32) )
  ChainSecret      ::= OCTET STRING  (SIZE (16..64))

  --
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
-- Block Definitions
--
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
PrefixBlock ::= SEQUENCE {
  version        [ 0 ] INTEGER OPTIONAL,
  key            [ 2 ] SymmetricKey
}

InnerMessageBlock ::= SEQUENCE {
  padding   OCTET STRING,
  prefix    CHOICE {
      plain          [ 1 1 0 1 1 ] PrefixBlock,
      -- contains prefix encrypted with receivers
      -- public key
      encrypted      [ 1 1 0 1 2 ] OCTET STRING
  },
  header  CHOICE  {
    -- debug/internal use only
    plain     [ 1 1 0 2 1 ] HeaderBlock,
    -- contains encrypted identity block
    encyrpted [ 1 1 0 2 2 ] OCTET STRING
  },
  -- contains signature of Identity [as stored in
  -- HeaderBlock; signed unencrypted HeaderBlock without
  -- Tag]
  identitySignature OCTET STRING,
  -- contains routing information (next hop) for the
  -- payloads
  routing     [ 1 1 0 0 1 ] CHOICE {
    plain     [ 1 1 0 3 1 ] RoutingBlock,
    -- contains encrypted routing block
    encyrpted [ 1 1 0 3 2 ] OCTET STRING
  },
  -- contains the actual payload
  payload     SEQUENCE (SIZE ( 0 ..maxPayloadBlks))
          OF OCTET STRING
}

HeaderBlock ::= SEQUENCE {
  -- Public key of the identity representing this
  -- transmission
  identityKey       AsymmetricKey,
  -- serial identifying this block
  serial            INTEGER ( 0 ..maxSerial),
  -- number of times this block may be replayed
  -- (Tuple is identityKey, serial while
  -- UsagePeriod of block)
  maxReplays        INTEGER ( 0 ..maxNumOfReplays),
  -- subsequent Blocks are not processed before
  -- valid time.
  -- Host may reject too long retention.
  -- Recomended validity support >= 1 Mt.
```

```
  valid            UsagePeriod,
  --  contains the MAC-Algorithm used for signing
  signAlgorithm     MacAlgorithmSpec,
  -- contains administrative requests such as
  -- quota requests
  requests          SEQUENCE
                  (SIZE (0..maxNumOfRequests))
                  OF HeaderRequest ,
  -- Reply Block for the requests
  requestReplyBlock   RoutingCombo OPTIONAL,
  -- padding and identitifier required to solve
  -- the cryptopuzzle
  identifier [12201] PuzzleIdentifier OPTIONAL,
  -- This is for solving crypto puzzles
  proofOfWork[12202] OCTET STRING OPTIONAL
}

RoutingBlock ::= SEQUENCE {
  -- contains the routingCombos
  routing    [331] SEQUENCE
                   (SIZE (0..maxRoutingBlks))
                 OF RoutingCombo,
  -- contains the mapping operations to map
  -- payloads to the workspace
  mappings   [332] SEQUENCE
                 (SIZE (0..maxPayloadBlks))
                 OF MapBlockOperation,
  -- contains a routing block which may be used
  -- when sending error messages back to the quota
  -- owner this routing block may be cached for
     -- future use
  replyBlock [332]   SEQUENCE {
      murb          RoutingCombo,
      maxReplay      INTEGER,
      validity       UsagePeriod
  } OPTIONAL
}

RoutingCombo ::= SEQUENCE {
  -- contains the period when the payload should
  -- be processed.
  -- Router might refuse too long queue retention
  -- Recommended support for retention >= 1h
  minProcessTime INTEGER
                  (0..maxDurationOfProcessing),
  maxProcessTime INTEGER
                  (0..maxDurationOfProcessing),
  -- The message key to encrypt the message
  peerKey        [401] SEQUENCE
                      (SIZE (1..maxNumOfReplays))
                   OF SymmetricKey OPTIONAL,
  -- contains the next recipient
  recipient      [402] BlendingSpec,
  -- PrefixBlock encrypted with message key
  mPrefix        [403] SEQUENCE
```

```
                          (SIZE (1..maxNumOfReplays))
                        OF OCTET STRING OPTIONAL,
     -- PrefixBlock encrypted with sender key
     cPrefix       [404] OCTET STRING OPTIONAL,
     -- HeaderBlock encrypted with sender key
     header        [405] OCTET STRING OPTIONAL,
     -- RoutingBlock encrypted with sender key
     routing       [406] OCTET STRING OPTIONAL,
     -- contains information for building messages
     -- (when used as MURB)
     -- ID 0 denotes original/local message
     -- ID 1-maxPayloadBlks denotes target message
       -- ID 32767 denotes a solicited reply block
     -- 32768-maxWorkspaceId shared workspace for all
     -- blocks of this identity)
     assembly      [407] SEQUENCE
                 (SIZE (0..maxAssemblyInstr))
                 OF PayloadOperation,
     -- optional for storage of the arrival time
     validity      [408] UsagePeriod OPTIONAL
   }

 END
```

A.2. The MessageVortex Ciphers Structures

```
MessageVortex-Ciphers DEFINITIONS EXPLICIT TAGS ::=
BEGIN
  EXPORTS SymmetricKey, AsymmetricKey, MacAlgorithmSpec,
          MacAlgorithm, CipherSpec, PRNGType;

  CipherSpec ::= SEQUENCE {
      asymmetric  [16001] AsymAlgSpec OPTIONAL,
      symmetric   [16002] SymAlgSpec OPTIONAL,
      mac         [16003] MacAlgorithmSpec OPTIONAL,
      cipherUsage [16004] CipherUsage
  }

  CipherUsage ::= ENUMERATED {
    sign          (200),
    encrypt       (210)
  }

  SymAlgSpec  ::=  SEQUENCE  {
    algorithm     [16101]SymmetricAlgorithm,
    -- if ommited: pkcs7
    padding       [16102]CipherPadding OPTIONAL,
    -- if ommited: cbc
    mode          [16103]CipherMode OPTIONAL,
    parameter     [16104]AlgParameters OPTIONAL
  }

  AsymAlgSpec  ::=  SEQUENCE  {
    algorithm     AsymmetricAlgorithm,
    -- if ommited: pkcs1
    padding       [16102]CipherPadding OPTIONAL,
    parameter     AlgParameters OPTIONAL
  }

  SymmetricKey ::= SEQUENCE {
    keyType       SymmetricAlgorithm,
    parameter     AlgParameters,
    key           OCTET STRING (SIZE(16..512))
  }

  AsymmetricKey ::= SEQUENCE {
    keyType       AsymmetricAlgorithm,
    -- private key encoded as PKCS#8/PrivateKeyInfo
    publicKey     [2] OCTET STRING,
    -- private key encoded as
       -- X.509/SubjectPublicKeyInfo
    privateKey    [3] OCTET STRING OPTIONAL
  }

  SymmetricAlgorithm ::= ENUMERATED {
    aes128         (1000),  -- required
    aes192         (1001),  -- optional support
    aes256         (1002),  -- required
    camellia128    (1100),  -- required
    camellia192    (1101),  -- optional support
    camellia256    (1102),  -- required
```

```
  twofish128    (1200),  -- optional support
  twofish192    (1201),  -- optional support
  twofish256    (1202)   -- optional support
}

AsymmetricAlgorithm ::= ENUMERATED {
  rsa         (2000),
  dsa         (2100),
  ec          (2200),
  ntru        (2300)
}
ECCurveType ::= ENUMERATED{
  secp384r1     (2500),
  sect409k1     (2501),
  secp521r1     (2502)
}
AlgParameters ::= SEQUENCE {
  keySize     [9000] INTEGER (0..65535) OPTIONAL,
  curveType   [9001] ECCurveType   OPTIONAL,
  iv          [9002] OCTET STRING  OPTIONAL,
  nonce       [9003] OCTET STRING  OPTIONAL,
  mode        [9004] CipherMode    OPTIONAL,
  padding     [9005] CipherPadding OPTIONAL,
  n           [9010] INTEGER       OPTIONAL,
  p           [9011] INTEGER       OPTIONAL,
  q           [9012] INTEGER       OPTIONAL,
  k           [9013] INTEGER       OPTIONAL,
  t           [9014] INTEGER       OPTIONAL
}

CipherMode ::= ENUMERATED {
  cbc         (10000), -- required
  ctr         (10001), -- required
  ccm         (10002), -- optional support
  gcm         (10003), -- optional support
  ocb         (10004), -- optional support
  ofb         (10005), -- optional support
  xts         (10006), -- optional support
  none        (10100)  -- required
}

CipherPadding ::= ENUMERATED {
  none         (10200), -- required
  pkcs1        (10201), -- required
  rsaesOaep    (10202), -- optional support
  oaepSha256Mgf1 (10203), -- optional support
  pkcs7        (10301), -- required
  ap           (10221)  -- required
}

MacAlgorithm ::= ENUMERATED {
  sha3-256      (3000), -- required
  sha3-384      (3001), -- optional support
  sha3-512      (3002), -- required
  ripemd160     (3100), -- optional support
  ripemd256     (3101), -- required
  ripemd320     (3102)  -- optional support
```

```
  }

  MacAlgorithmSpec  ::=  SEQUENCE  {
    algorithm     MacAlgorithm,
    parameter     AlgParameters
  }

  PRNGAlgorithmSpec ::= SEQUENCE {
    type          PRNGType,
    seed          OCTET STRING
  }

  PRNGType ::= ENUMERATED {
    mrg32k3a      (10300), -- required
    blumMicali    (10301)  -- required
  }

END
```

A.3. The MessageVortex Request Structures

```
MessageVortex-Requests DEFINITIONS EXPLICIT TAGS ::=
BEGIN
  EXPORTS HeaderRequest;
  IMPORTS RequirementBlock
              FROM MessageVortex-Requirements
          UsagePeriod, NodeSpec
              FROM MessageVortex-Helpers;

  HeaderRequest ::= CHOICE {
    identity      [ 0 ] HeaderRequestIdentity,
    capabilities  [ 1 ] HeaderRequestCapability,
    messageQuota  [ 2 ] HeaderRequestIncreaseMessageQuota,
    transferQuota [ 3 ] HeaderRequestIncreaseTransferQuota,
    quotaQuery    [ 4 ] HeaderRequestQuota,
    nodeQuery     [ 5 ] HeaderRequestNodes,
    replace       [ 6 ] HeaderRequestReplaceIdentity
  }

  HeaderRequestIdentity ::= SEQUENCE {
    period UsagePeriod
  }

  HeaderRequestReplaceIdentity ::= SEQUENCE {
    replace           SEQUENCE {
        old           NodeSpec,
        new           NodeSpec OPTIONAL
    },
    identitySignature OCTET STRING
  }

  HeaderRequestQuota ::= SEQUENCE {
  }

  HeaderRequestNodes ::= SEQUENCE {
    numberOfNodes   INTEGER (0..255)
  }

  HeaderRequestIncreaseMessageQuota ::= SEQUENCE {
    messages INTEGER (0..4294967295)
  }

  HeaderRequestIncreaseTransferQuota ::= SEQUENCE {
    size     INTEGER (0..4294967295)
  }

  HeaderRequestCapability ::= SEQUENCE {
    period UsagePeriod
  }

  HeaderRequestUpgrade ::= SEQUENCE {
    version     OCTET STRING,
    identifier  OCTET STRING
  }
```

```
END
```

A.4. The MessageVortex Replies Structures

```
MessageVortex-Replies DEFINITIONS EXPLICIT TAGS ::=
BEGIN
 EXPORTS SpecialBlock;
 IMPORTS BlendingSpec, NodeSpec
             FROM MessageVortex-Helpers
         RequirementBlock
             FROM MessageVortex-Requirements
         CipherSpec, PRNGType, MacAlgorithm
             FROM MessageVortex-Ciphers
         maxGFSize
             FROM MessageVortex-Operations
         maxNumberOfReplays
             FROM MessageVortex-Schema;

 SpecialBlock ::= CHOICE {
   capabilities  [1]  ReplyCapability,
   requirement   [2]  SEQUENCE (SIZE (1..127))
                   OF RequirementBlock,
   quota         [4]  ReplyCurrentQuota,
   nodes         [5]  ReplyNodes,
   status        [99] StatusBlock
 }

 StatusBlock ::= SEQUENCE {
   code          StatusCode
 }

 StatusCode ::= ENUMERATED {

   -- System messages
   ok                          (2000),
   quotaStatus                 (2101),
   puzzleRequired              (2201),

   -- protocol usage failures
   transferQuotaExceeded       (3001),
   messageQuotaExceeded        (3002),
   requestedQuotaOutOfBand     (3003),
   identityUnknown             (3101),
   messageChunkMissing         (3201),
   messageLifeExpired          (3202),
   puzzleUnknown               (3301),

   -- capability errors
   macAlgorithmUnknown         (3801),
   symmetricAlgorithmUnknown   (3802),
   asymmetricAlgorithmUnknown  (3803),
   prngAlgorithmUnknown        (3804),
   missingParameters           (3820),
   badParameters               (3821),

   -- Mayor host specific errors
   hostError                   (5001)
 }
```

```
ReplyNodes ::= SEQUENCE {
  node   SEQUENCE (SIZE (1..5))
       OF NodeSpec
}

ReplyCapability ::= SEQUENCE {
  -- supported ciphers
  cipher           SEQUENCE (SIZE (2..256))
                   OF CipherSpec,
  -- supported mac algorithms
  mac              SEQUENCE (SIZE (2..256))
                   OF MacAlgorithm,
  -- supported PRNGs
  prng             SEQUENCE (SIZE (2..256))
                   OF PRNGType,
  -- maximum number of bytes to be transferred
    -- (outgoing bytes in vortex message without blending)
  maxTransferQuota  INTEGER (0..4294967295),
  -- maximum number of messages to process for this identity
  maxMessageQuota   INTEGER (0..4294967295),
  -- maximum simultaneously tracked header serials
  maxHeaderSerials  INTEGER (0..4294967295),
  -- maximum simultaneously valid build operations in workspace
  maxBuildOps       INTEGER (0..4294967295),
  -- maximum payload size
  maxPayloadSize    INTEGER (0..4294967295),
  -- maximum active payloads (without intermediate products)
  maxActivePayloads INTEGER (0..4294967295),
  -- maximum header lifespan in seconds
  maxHeaderLive     INTEGER (0..4294967295),
  -- maximum number of replays accepted,
  maxReplay         INTEGER (0..maxNumberOfReplays),
  -- Supported inbound blending
  supportedBlendingIn SEQUENCE OF BlendingSpec,
  -- Supported outbound blending
  supportedBlendingOut SEQUENCE OF BlendingSpec,
  -- supported galoise fields
  supportedGFSize   SEQUENCE OF INTEGER (1..maxGF)
}

ReplyCurrentQuota ::= SEQUENCE {
  messages INTEGER (0..4294967295),
  size     INTEGER (0..4294967295)
}

ReplyUpgrade ::= SEQUENCE {
  -- The offered version
  version    [0] OCTET STRING,
  -- The offered identitfier
  identifier [1] OCTET STRING,
  -- The archive or blob containing the software
  blob       [2] OCTET STRING OPTIONAL
}
```

```
END
```

A.5. The MessageVortex Requirements Structures

```
MessageVortex-Requirements DEFINITIONS EXPLICIT TAGS ::=
BEGIN
  EXPORTS RequirementBlock;
  IMPORTS MacAlgorithmSpec
              FROM MessageVortex-Ciphers
          UsagePeriod, UsagePeriod
              FROM MessageVortex-Helpers;

  RequirementBlock ::= CHOICE {
    puzzle  [ 1 ] RequirementPuzzleRequired,
    payment [ 2 ] RequirementPaymentRequired
  }

  RequirementPuzzleRequired ::= SEQUENCE {
    -- bit sequence at beginning of hash from
       -- the encrypted identity block
    challenge     BIT STRING,
    mac           MacAlgorithmSpec,
    valid         UsagePeriod,
    identifier    INTEGER (0..4294967295)
  }

  RequirementPaymentRequired ::= SEQUENCE {
    account       OCTET STRING,
    ammount       REAL,
    currency      Currency
  }

  Currency ::= ENUMERATED {
    btc     (8001),
    eth     (8002),
    zec     (8003)
  }

END
```

A.6. The MessageVortex Helpers Structures

```
MessageVortex-Helpers DEFINITIONS EXPLICIT TAGS ::=
BEGIN
  EXPORTS UsagePeriod, BlendingSpec, NodeSpec;
  IMPORTS AsymmetricKey, SymmetricKey
             FROM MessageVortex-Ciphers;

  -- the maximum number of embeddable parameters
  maxNumberOfParameter      INTEGER ::= 127

  UsagePeriod ::= CHOICE {
    absolute [2] AbsoluteUsagePeriod,
    relative [3] RelativeUsagePeriod
  }

  AbsoluteUsagePeriod ::= SEQUENCE {
    notBefore       [0]     GeneralizedTime OPTIONAL,
    notAfter        [1]     GeneralizedTime OPTIONAL
  }

  RelativeUsagePeriod ::= SEQUENCE {
    notBefore       [0]     INTEGER OPTIONAL,
    notAfter        [1]     INTEGER OPTIONAL
  }

  -- contains a node spec of a routing point
  -- At the moment either smtp:<email> or xmpp:<jabber>
  BlendingSpec ::= SEQUENCE {
    target          [1] NodeSpec,
    blendingType    [2] IA5String,
    parameter       [3] SEQUENCE
                    ( SIZE (0..maxNumberOfParameter) )
                    OF BlendingParameter
  }

  BlendingParameter ::= CHOICE {
    offset          [1] INTEGER,
    symmetricKey    [2] SymmetricKey,
    asymmetricKey   [3] AsymmetricKey,
    passphrase      [4] OCTET STRING
  }

  NodeSpec ::= SEQUENCE {
    transportProtocol [1] Protocol,
    recipientAddress  [2] IA5String,
    recipientKey      [3] AsymmetricKey OPTIONAL
  }

  Protocol ::= ENUMERATED {
    smtp  (100),
    xmmp  (110)
  }

END
```

A.7. The MessageVortex Additional Structures

```
-- States reflected:
--   Tuple()=Val()[vallidity; allowed operations]
--   {Store}
-- - Tuple(identity)=Val(messageQuota,transferQuota,
--   sequence of Routingblocks for Error Message
--   Routing) [validity; Requested at creation; may
--   be extended upon request] {identityStore}
-- - Tuple(Identity,Serial)=maxReplays ['valid' from
--   Identity Block; from First Identity Block; may
--   only be reduced] {IdentityReplayStore}

MessageVortex-NonProtocolBlocks DEFINITIONS
                         EXPLICIT TAGS ::=
BEGIN
  IMPORTS PrefixBlock, InnerMessageBlock,
          RoutingBlock,
          maxWorkspaceID
               FROM MessageVortex-Schema
          UsagePeriod, NodeSpec, BlendingSpec
               FROM MessageVortex-Helpers
          AsymmetricKey
               FROM MessageVortex-Ciphers
          RequirementBlock
               FROM MessageVortex-Requirements;

  -- maximum size of transfer quota in bytes of an
  -- identity
  maxTransferQuota         INTEGER ::= 4294967295
  -- maximum # of messages quota in messages of an
  -- identity
  maxMessageQuota          INTEGER ::= 4294967295

  -- do not use these blocks for protocol encoding
  -- (internal only)
  VortexMessage ::= SEQUENCE {
    prefix        CHOICE {
      plain          [10011] PrefixBlock,
      -- contains prefix encrypted with receivers
             -- public key
      encrypted      [10012] OCTET STRING
    },
    innerMessage  CHOICE {
      plain     [10021] InnerMessageBlock,
      -- contains inner message encrypted with
      -- Symmetric key from prefix
      encrypted [10022] OCTET STRING
    }
  }

  MemoryPayloadChunk ::= SEQUENCE {
    id           INTEGER (0..maxWorkspaceID),
    payload       [100] OCTET STRING,
    validity      UsagePeriod
```

```
  }

  IdentityStore ::= SEQUENCE {
    identities SEQUENCE (SIZE (0..4294967295))
            OF IdentityStoreBlock
  }

  IdentityStoreBlock ::= SEQUENCE {
    valid         UsagePeriod,
    messageQuota   INTEGER (0..maxMessageQuota),
    transferQuota  INTEGER (0..maxTransferQuota),
    -- if omitted this is a node identity
    identity       [1001] AsymmetricKey OPTIONAL,
    -- if ommited own identity key
    nodeAddress    [1002] NodeSpec      OPTIONAL,
    -- Contains the identity of the owning node;
    -- May be ommited if local node
    nodeKey        [1003] SEQUENCE OF AsymmetricKey
                          OPTIONAL,
    routingBlocks  [1004] SEQUENCE OF RoutingBlock
                          OPTIONAL,
    replayStore    [1005] IdentityReplayStore,
    requirement    [1006] RequirementBlock OPTIONAL
  }

  IdentityReplayStore ::= SEQUENCE {
    replays   SEQUENCE (SIZE (0..4294967295))
           OF IdentityReplayBlock
  }

  IdentityReplayBlock ::= SEQUENCE {
    identity        AsymmetricKey,
    valid           UsagePeriod,
    replaysRemaining INTEGER (0..4294967295)
  }

END
```

Appendix B. Changelog

Version #	Date	Changes
0	11-2018	Initial version
1	02-2019	Removed term block. Added more precise spec about blending. Change in spec for XMPP blending (from XEP-234 to XEP-231). Restructured ASN.1.

Version #	Date	Changes
2	03-2019	Language and consistency improvments. Added example for chunnked plain embedding. Added pseudocode for incomming message processing. Improved wording of hashes in ASN.1.
3	09-2019	Removed LaTeX notation in padding.
4	03-2020	Added spec for Software update using MV. Minor language improvments.
5	09-2020	Reinserted lost ASN.1 specs (unintentinally lost in last two versions). Added changelog. Modified padding to improve credibility of bad values.

Table 1: changes in versions

Author's Address

Martin Gwerder
University of Applied Sciences and Arts Northwestern Switzerland
Bahnhofstrasse 5
CH-5210 Windisch
Switzerland
Phone: +41 56 202 76 81
Email: rfc@messagevortex.net