Sphinx Cryptographic Packet Format

Yawning Angel
George Danezis
Claudia Diaz
Ania Piotrowska
David Stainton

Table of Contents

Abstract

This document defines the Sphinx cryptographic packet format for decryption mix networks, and provides a parameterization based around generic cryptographic primitives types. This document does not introduce any new crypto, but is meant to serve as an implementation guide.

1. Introduction

The Sphinx cryptographic packet format is a compact and provably secure design introduced by George Danezis and Ian Goldberg SPHINX09. It supports a full set of security features: indistinguishable replies, hiding the path length and relay position, detection of tagging attacks and replay attacks, as well as providing unlinkability for each leg of the packet's journey over the network.

1.1 Terminology

- `Message` - A variable-length sequence of octets sent anonymously through the network.
- `Packet` - A fixed-length sequence of octets transmitted anonymously through the network, containing the encrypted message and metadata for routing.
- `Header` - The packet header consisting of several components, which convey the information necessary to verify packet integrity and correctly process the packet.
- `Payload` - The fixed-length portion of a packet containing an encrypted message or part of a message, to be delivered anonymously.
- `Group` - A finite set of elements and a binary operation that satisfy the properties of closure, associativity, invertability, and the presence of an identity element.
- `Group element` - An individual element of the group.
- `Group generator` - A group element capable of generating any other element of the group, via repeated applications of the generator and the group operation.

1.2 Conventions Used in This Document

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.

The “C” style Presentation Language as described in RFC5246 Section 4 is used to represent data structures, except for cryptographic attributes, which are specified as opaque byte vectors.

- `x | y` denotes the concatenation of x and y.
- `x ^ y` denotes the bitwise XOR of x and y.
- `byte` an 8-bit octet.
- `x[a:b]` denotes the sub-vector of x where a/b denote the start/end byte indexes (inclusive-exclusive); a/b may be omitted to signify the start/end of the vector x respectively.
- `x[y]` denotes the y'th element of list x.
- `x.len` denotes the length of list x.
- `ZEROBYTES(N)` denotes N bytes of 0x00.
- `RNG(N)` denotes N bytes of cryptographic random data.
- `LEN(N)` denotes the length in bytes of N.
- `CONSTANT_TIME_CMP(x, y)` denotes a constant time comparison between the byte vectors x and y, returning true iff x and y are equal.

2. Cryptographic Primitives

This specification uses the following cryptographic primitives as the foundational building blocks for Sphinx:

- `H(M)` - A cryptographic hash function which takes an octet array M to produce a digest consisting of a `HASH_LENGTH` byte octet array. `H(M)` MUST be pre-image and collision resistant.

- `MAC(K, M)` - A cryptographic message authentication code function which takes a `M_KEY_LENGTH` byte octet array key `K` and arbitrary length octet array message `M` to produce an authentication tag consisting of a `MAC_LENGTH` byte octet array.

- `KDF(SALT, IKM)` - A key derivation function which takes an arbitrary length octet array salt `SALT` and an arbitrary length octet array initial key `IKM`, to produce an octet array of arbitrary length.

- `S(K, IV)` - A pseudo-random generator (stream cipher) which takes a `S_KEY_LENGTH` byte octet array key `K` and a `S_IV_LENGTH` byte octet array initialization vector `IV` to produce an octet array key stream of arbitrary length.

- `SPRP_Encrypt(K, M)/SPRP_Decrypt(K, M)` - A strong pseudo-random permutation (SPRP) which takes a `SPRP_KEY_LENGTH` byte octet array key `K` and arbitrary length message `M`, and produces the encrypted ciphertext or decrypted plaintext respectively.

  When used with the default payload authentication mechanism, the SPRP MUST be "fragile" in that any amount of modifications to `M` results in a large number of unpredictable changes across the whole message upon a `SPRP_Encrypt()` or `SPRP_Decrypt()` operation.

- `EXP(X, Y)` - An exponentiation function which takes the `GROUP_ELEMENT_LENGTH` byte octet array group elements `X` and `Y`, and returns `X ^^ Y` as a `GROUP_ELEMENT_LENGTH` byte octet array.

  Let `G` denote the generator of the group, and `EXP_KEYGEN()` return a `GROUP_ELEMENT_LENGTH` byte octet array group element usable as private key.

  The group defined by `G` and `EXP(X, Y)` MUST satisfy the Decision Diffie-Hellman problem.

- `EXP_KEYGEN()` - Returns a new "suitable" private key for `EXP()`.

2.1 Sphinx Key Derivation Function

Sphinx Packet creation and processing uses a common Key Derivation Function (KDF) to derive the required MAC and symmetric cryptographic keys from a per-hop shared secret.

The output of the KDF is partitioned according to the following structure:

```
struct {
    opaque header_mac[M_KEY_LENGTH];
    opaque header_encryption[S_KEY_LENGTH];
    opaque header_encryption_iv[S_IV_LENGTH];
    opaque payload_encryption[SPRP_KEY_LENGTH]
    opaque blinding_factor[GROUP_ELEMENT_LENGTH];
} SphinxPacketKeys;

Sphinx_KDF( info, shared_secret ) -> packet_keys
```

Inputs:

- `info` The optional context and application specific information.
- `shared_secret` The per-hop shared secret derived from the Diffie-Hellman key exchange.

Outputs:

- `packet_keys` The SphinxPacketKeys required to handle packet creation or processing.

The output packet_keys is calculated as follows:

```
kdf_out = KDF( info, shared_secret )
packet_keys = kdf_out[:LEN( SphinxPacketKeys )]
```

3. Sphinx Packet Parameters

3.1 Sphinx Parameter Constants

The Sphinx Packet Format is parameterized by the implementation based on the application and security requirements.

- AD_LENGTH - The constant amount of per-packet unencrypted additional data in bytes.
- PAYLOAD_TAG_LENGTH - The length of the message payload authentication tag in bytes. This SHOULD be set to at least 16 bytes (128 bits).
- PER_HOP_RI_LENGTH - The length of the per-hop Routing Information (Section 4.1.1 <4.1.1>) in bytes.
- NODE_ID_LENGTH - The node identifier length in bytes.
- RECIPIENT_ID_LENGTH - The recipient identifier length in bytes.
- SURB_ID_LENGTH - The Single Use Reply Block (Section 7 <7.0>) identifier length in bytes.
- MAX_HOPS - The maximum number of hops a packet can traverse.
- PAYLOAD_LENGTH - The per-packet message payload length in bytes, including a PAYLOAD_TAG_LENGTH byte authentication tag.
- KDF_INFO - A constant opaque byte vector used as the info parameter to the KDF for the purpose of domain separation.

3.2 Sphinx Packet Geometry

The Sphinx Packet Geometry is derived from the Sphinx Parameter Constants Section 3.1. These are all derived parameters, and are primarily of interest to implementors.

- ROUTING_INFO_LENGTH - The total length of the "routing information" Sphinx Packet Header component in bytes:

```
ROUTING_INFO_LENGTH = PER_HOP_RI_LENGTH * MAX_HOPS
```

- HEADER_LENGTH - The length of the Sphinx Packet Header in bytes:

```
HEADER_LENGTH = AD_LENGTH + GROUP_ELEMENT_LENGTH + ROUTING_INFO_LENGTH + MAC_LENGT
```

- PACKET_LENGTH - The length of the Sphinx Packet in bytes:

```
PACKET_LENGTH = HEADER_LENGTH + PAYLOAD_LENGTH
```

4. The Sphinx Cryptographic Packet Structure

Each Sphinx Packet consists of two parts: the Sphinx Packet Header and the Sphinx Packet Payload:

```
struct {
    opaque header[HEADER_LENGTH];
    opaque payload[PAYLOAD_LENGTH];
} SphinxPacket;
```

- `header` - The packet header consists of several components, which convey the information necessary to verify packet integrity and correctly process the packet.
- `payload` - The application message data.

4.1 Sphinx Packet Header

The Sphinx Packet Header refers to the block of data immediately preceding the Sphinx Packet Payload in a Sphinx Packet.

The structure of the Sphinx Packet Header is defined as follows:

```
struct {
    opaque additional_data[AD_LENGTH]; /* Unencrypted. */
    opaque group_element[GROUP_ELEMENT_LENGTH];
    opaque routing_information[ROUTING_INFO_LENGTH];
    opaque MAC[MAC_LENGTH];
} SphinxHeader;
```

- `additional_data` - Unencrypted per-packet Additional Data (AD) that is visible to every hop. The AD is authenticated on a per-hop basis.

  As the additional_data is sent in the clear and traverses the network unaltered, implementations MUST take care to ensure that the field cannot be used to track individual packets.

- `group_element` - An element of the cyclic group, used to derive the per-hop key material required to authenticate and process the rest of the SphinxHeader and decrypt a single layer of the Sphinx Packet Payload encryption.

- `routing_information` - A vector of per-hop routing information, encrypted and authenticated in a nested manner. Each element of the vector consists of a series of routing commands, specifying all of the information required to process the packet.

  The precise encoding format is specified in `Section 4.1.1 <4.1.1>`.

- `MAC` - A message authentication code tag covering the additional_data, group_element, and routing_information.

4.1.1 Per-hop routing information

The routing_information component of the Sphinx Packet Header contains a vector of per-hop routing information. When processing a packet, the per hop processing is set up such that the first element in the vector contains the routing commands for the current hop.

The structure of the routing information is as follows:

```
struct {
    RoutingCommand routing_commands<1..2^8-1>; /* PER_HOP_RI_LENGTH bytes */
    opaque encrypted_routing_commands[ROUTING_INFO_LENGTH - PER_HOP_RI_LENGTH];
} RoutingInformation;
```

The structure of a single routing command is as follows:

```
struct {
```

```
    RoutingCommandType command;
    select (RoutingCommandType) {
        case null:               NullCommand;
        case next_node_hop:      NextNodeHopCommand;
        case recipient:          RecipientCommand;
        case surb_reply:         SURBReplyCommand;
    };
} RoutingCommand;
```

The following routing commands are currently defined:

```
enum {
    null(0),
    next_node_hop(1),
    recipient(2),
    surb_reply(3),

    /* Routing commands between 0 and 0x7f are reserved. */

    (255)
} RoutingCommandType;
```

The null routing command structure is as follows:

```
struct {
    opaque padding<0..PER_HOP_RI_LENGTH-1>;
} NullCommand;
```

The next_node_hop command structure is as follows:

```
struct {
    opaque next_hop[NODE_ID_LENGTH];
    opaque MAC[MAC_LENGTH];
} NextNodeHopCommand;
```

The recipient command structure is as follows:

```
struct {
    opaque recipient[RECIPEINT_ID_LENGTH];
} RecipientCommand;
```

The surb_reply command structure is as follows:

```
struct {
    opaque id[SURB_ID_LENGTH];
} SURBReplyCommand;
```

While the `NullCommand` padding field is specified as opaque, implementations SHOULD zero fill the padding. The choice of `0x00` as the terminal NullCommand is deliberate to ease implementation, as `ZEROBYTES(N)` produces a valid NullCommand RoutingCommand, resulting in “appending zero filled padding” producing valid output.

Implementations MUST pad the routing_commands vector so that it is exactly `PER_HOP_RI_LENGTH` bytes, by appending a terminal NullCommand if necessary.

Every non-terminal hop's `routing_commands` MUST include a `NextNodeHopCommand`.

4.2 Sphinx Packet Payload

The Sphinx Packet Payload refers to the block of data immediately following the Sphinx Packet Header in a Sphinx Packet.

For most purposes the structure of the Sphinx Packet Payload can be treated as a single contiguous byte vector of opaque data.

Upon packet creation, the payload is repeatedly encrypted (unless it is a SURB Reply, see `Section 7.0` via keys derived from the Diffie-Hellman key exchange between the packet's `group_element` and the public key of each node in the path.

Authentication of packet integrity is done by prepending a tag set to a known value to the plaintext prior to the first encrypt operation. By virtue of the fragile nature of the SPRP function, any alteration to the encrypted payload as it traverses the network will result in an irrecoverably corrupted plaintext when the payload is decrypted by the recipient.

5. Sphinx Packet Creation

For the sake of brevity, the pseudocode for all of the operations will take a vector of the following PathHop structure as a parameter named path[] to specify the path a packet will traverse, along with the per-hop routing commands and per-hop public keys.

```
struct {
    /* There is no need for a node_id here, as
       routing_commands[0].next_hop specifies that
       information for all non-terminal hops. */
    opaque public_key[GROUP_ELEMENT_LENGTH];
    RoutingCommand routing_commands<1...2^8-1>;
} PathHop;
```

It is assumed that each routing_commands vector except for the terminal entry contains at least a RoutingCommand consisting of a partially assembled NextNodeHopCommand with the `next_hop` element filled in with the identifier of the next hop.

5.1 Create a Sphinx Packet Header

Both the creation of a Sphinx Packet and the creation of a SURB requires the generation of a Sphinx Packet Header, so it is specified as a distinct operation.

```
Sphinx_Create_Header( additional_data, path[] ) -> sphinx_header,
                                                   payload_keys
```

Inputs:

- `additional_data` The Additional Data that is visible to every node along the path in the header.

- `path` The vector of PathHop structures in hop order, specifying the node id, public key, and routing commands for each hop.

Outputs: `sphinx_header` The resulting Sphinx Packet Header.

- `payload_keys` The vector of SPRP keys used to encrypt the Sphinx Packet Payload, in hop order.

The `Sphinx_Create_Header` operation consists of the following steps:

1. Derive the key material for each hop.

```
num_hops = route.len
route_keys = [ ]
route_group_elements = [ ]
priv_key = EXP_KEYGEN()

/* Calculate the key material for the 0th hop. */
group_element = EXP( G, priv_key )
route_group_elements += group_element
shared_secret = EXP( path[0].public_key, priv_key )
route_keys += Sphinx_KDF( KDF_INFO, shared_secret )
blinding_factor = keys[0].blinding_factor

/* Calculate the key material for rest of the hops. */
for i = 1; i < num_hops; ++i:
    shared_secret = EXP( path[i].public_key, priv_key )
    for j = 0; j < i; ++j:
        shared_secret = EXP( shared_secret, keys[j].blinding_factor )
    route_keys += Sphinx_KDF( KDF_INFO, shared_secret )
    group_element = EXP( group_element, keys[i-1].blinding_factor )
    route_group_elements += group_element
```

At the conclusion of the derivation process:

- `route_keys` - A vector of per-hop SphinxKeys.
- `route_group_elements` - A vector of per-hop group elements.

2. Derive the routing_information keystream and encrypted padding for each hop.

```
ri_keystream = [ ]
ri_padding = [ ]

for i = 0; i < num_hops; ++i:
    keystream = ZEROBYTES( ROUTING_INFO_LENGTH + PER_HOP_RI_LENGTH ) ^
                  S( route_keys[i].header_encryption,
                     route_keys[i].header_encryption_iv )
    ks_len = LEN( keystream ) - (i + 1) * PER_HOP_RI_LENGTH

    padding = keystream[ks_len:]
    if i > 0:
        prev_pad_len = LEN( ri_padding[i-1] )
        padding = padding[:prev_pad_len] ^ ri_padding[i-1] |
            padding[prev_pad_len]
```

```
    ri_keystream += keystream[:ks_len]
    ri_padding += padding

At the conclusion of the derivation process:
   ri_keystream - A vector of per-hop routing_information
                  encryption keystreams.
   ri_padding   - The per-hop encrypted routing_information
                  padding.
```

3. Create the routing_information block.

```
/* Start with the terminal hop, and work backwards. */
i = num_hops - 1

/* Encode the terminal hop's routing commands. As the
   terminal hop can never have a NextNodeHopCommand, there
   are no per-hop alterations to be made. */
ri_fragment = path[i].routing_commands |
   ZEROBYTES( PER_HOP_RI_LENGTH - LEN( path[i].routing_commands ) )

/* Encrypt and MAC. */
ri_fragment ^= ri_keystream[i]
mac = MAC( route_keys[i].header_mac, additional_data |
               route_group_elements[i] | ri_fragment |
               ri_padding[i-1] )
routing_info = ri_fragment
if num_hops < MAX_HOPS:
    pad_len = (MAX_HOPS - num_hops) * PER_HOP_RI_LENGTH
    routing_info = routing_info | RNG( pad_len )

/* Calculate the routing info for the rest of the hops. */
for i = num_hops - 2; i >= 0; --i:
    cmds_to_encode = [ ]

    /* Find and finalize the NextNodeHopCommand. */
    for j = 0; j < LEN( path[i].routing_commands; j++:
        cmd = path[i].routing_commands[j]
        if cmd.command == next_node_hop:
          /* Finalize the NextNodeHopCommand. */
          cmd.MAC = mac
        cmds_to_encode = cmds_to_encode + cmd /* Append */

    /* Append a terminal NullCommand. */
    ri_fragment = cmds_to_encode |
        ZEROBYTES( PER_HOP_RI_LENGTH - LEN( cmds_to_encode ) )

    /* Encrypt and MAC */
    routing_info = ri_fragment | routing_info /* Prepend. */
    routing_info ^= ri_keystream[i]
    if i > 0:
        mac = MAC( route_keys[i].header_mac, additional_data |
                   route_group_elements[i] | routing_info |
```

```
                    ri_padding[i-1] )
    else:
        mac = MAC( route_keys[i].header_mac, additional_data |
                    route_group_elements[i] | routing_info )

At the conclusion of the derivation process:
   routing_info - The completed routing_info block.
   mac          - The MAC for the 0th hop.
```

4. Assemble the completed Sphinx Packet Header and Sphinx Packet Payload SPRP key vector.

```
/* Assemble the completed Sphinx Packet Header. */
SphinxHeader sphinx_header
sphinx_header.additional_data = additional_data
sphinx_header.group_element = route_group_elements[0] /* From step 1. */
sphinx_header.routing_info = routing_info   /* From step 3. */
sphinx_header.mac = mac                     /* From step 3. */

/* Preserve the Sphinx Payload SPRP keys, to return to the
   caller. */
payload_keys = [ ]
for i = 0; i < nr_hops; ++i:
    payload_keys += route_keys[i].payload_encryption

At the conclusion of the assembly process:
   sphinx_header - The completed sphinx_header, to be returned.
   payload_keys  - The vector of SPRP keys, to be returned.
```

5.2 Create a Sphinx Packet

```
Sphinx_Create_Packet( additional_data, path[], payload ) -> sphinx_packet
```

Inputs:

- `additional_data` The Additional Data that is visible to every node along the path in the header.
- `path` The vector of PathHop structures in hop order, specifying the node id, public key, and routing commands for each hop.
- `payload` The packet payload message plaintext.

Outputs:

- `sphinx_packet` The resulting Sphinx Packet.

The `Sphinx_Create_Packet` operation consists of the following steps:

1. Create the Sphinx Packet Header and SPRP key vector.

```
sphinx_header, payload_keys =
    Sphinx_Create_Header( additional_data, path )
```

2. Prepend the authentication tag, and append padding to the payload.

```
payload = ZERO_BYTES( PAYLOAD_TAG_LENGTH ) | payload
payload = payload | ZERO_BYTES( PAYLOAD_LENGTH - LEN( payload ) )
```

3. Encrypt the payload.

```
for i = nr_hops - 1; i >= 0; --i:
    payload = SPRP_Encrypt( payload_keys[i], payload )
```

4. Assemble the completed Sphinx Packet.

```
SphinxPacket sphinx_packet
sphinx_packet.header = sphinx_header
sphinx_packet.payload = payload
```

6. Sphinx Packet Processing

Mix nodes process incoming packets first by performing the `Sphinx_Unwrap` operation to authenticate and decrypt the packet, and if applicable prepare the packet to be forwarded to the next node.

If `Sphinx_Unwrap` returns an error for any given packet, the packet MUST be discarded with no additional processing.

After a packet has been unwrapped successfully, a replay detection tag is checked to ensure that the packet has not been seen before. If the packet is a replay, the packet MUST be discarded with no additional processing.

The routing commands for the current hop are interpreted and executed, and finally the packet is forwarded to the next mix node over the network or presented to the application if the current node is the final recipient.

6.1 Sphinx_Unwrap Operation

The `Sphinx_Unwrap` operation is the majority of the per-hop packet processing, handling authentication, decryption, and modifying the packet prior to forwarding it to the next node.

```
Sphinx_Unwrap( routing_private_key, sphinx_packet ) -> sphinx_packet,
                                                      routing_commands,
                                                      replay_tag
```

Inputs:

- `private_routing_key` A group element GROUP_ELEMENT_LENGTH bytes in length, that serves as the unwrapping Mix's private key.
- `sphinx_packet` A Sphinx packet to unwrap.

Outputs:

- `error` Indicating a unsuccessful unwrap operation if applicable.
- `sphinx_packet` The resulting Sphinx packet.
- `routing_commands` A vector of RoutingCommand, specifying the post unwrap actions to be taken on the packet.
- `replay_tag` A tag used to detect whether this packet was processed before.

The `Sphinx_Unwrap` operation consists of the following steps:

0 (Optional) Examine the Sphinx Packet Header's Additional Data.

If the header's `additional_data` element contains information required to complete the unwrap operation, such as specifying the packet format version or the cryptographic primitives used examine it now.

Implementations MUST NOT treat the information in the `additional_data` element as trusted until after the completion of Step 3 ("Validate the Sphinx Packet Header").

1. Calculate the hop's shared secret, and replay_tag.

```
hdr = sphinx_packet.header
shared_secret = EXP( hdr.group_element, private_routing_key )
replay_tag = H( shared_secret )
```

2. Derive the various keys required for packet processing.

```
keys = Sphinx_KDF( KDF_INFO, shared_secret )
```

3. Validate the Sphinx Packet Header.

```
derived_mac = MAC( keys.header_mac, hdr.additional_data |
                  hdr.group_element |
                  hdr.routing_information )
if !CONSTANT_TIME_CMP( derived_mac, hdr.MAC):
    /* MUST abort processing if the header is invalid. */
    return ErrorInvalidHeader
```

4. Extract the per-hop routing commands for the current hop.

```
/* Append padding to preserve length-invariance, as the routing
    commands for the current hop will be removed. */
padding = ZEROBYTES( PER_HOP_RI_LENGTH )
B = hdr.routing_information | padding

/* Decrypt the entire routing_information block. */
B = B ^ S( keys.header_encryption, keys.header_encryption_iv )
```

5. Parse the per-hop routing commands.

```
cmd_buf = B[:PER_HOP_RI_LENGTH]
new_routing_information = B[PER_HOP_RI_LENGTH:]

next_mix_command_idx = -1
routing_commands = [ ]
for idx = 0; idx < PER_HOP_RI_LENGTH {
     /* WARNING: Bounds checking omitted for brevity. */
     cmd_type = b[idx]
     cmd = NULL
     switch cmd_type {
        case null: goto done  /* No further commands. */
```

```
            case next_node_hop:
                cmd = RoutingCommand( B[idx:idx+1+LEN( NextNodeHopCommand )] )
                next_mix_command_idx = i /* Save for step 7. */
                idx += 1 + LEN( NextNodeHopCommand )
                break

            case recipient:
                cmd = RoutingCommand( B[idx:idx+1+LEN( FinalDestinationCommand )] )
                idx += 1 + LEN( RecipientCommand )
                break

            case surb_reply:
                cmd = RoutingCommand( B[idx:idx+1+LEN( SURBReplyCommand )] )
                idx += 1 + LEN( SURBReplyCommand )
                break

          default:
                /* MUST abort processing on unrecognized commands. */
                return ErrorInvalidCommand
        }
        routing_commands += cmd /* Append cmd to the tail of the list. */
    }
    done:
```

At the conclusion of the parsing step:

- `routing_commands` - A vector of SphinxRoutingCommand, to be applied at this hop.
- `new_routing_information` - The routing_information block to be sent to the next hop if any.

6. Decrypt the Sphinx Packet Payload.

```
payload = sphinx_packet.payload
payload = SPRP_Decrypt( key.payload_encryption, payload )
sphinx_packet.payload = payload
```

7. Transform the packet for forwarding to the next mix, if the routing commands vector included a NextNodeHopCommand.

```
if next_mix_command_idx != -1:
    cmd = routing_commands[next_mix_command_idx]
    hdr.group_element = EXP( hdr.group_element, keys.blinding_factor )
    hdr.routing_information = new_routing_information
    hdr.mac = cmd.MAC
    sphinx_packet.hdr = hdr
```

6.2 Post Sphinx_Unwrap Processing

Upon the completion of the `Sphinx_Unwrap` operation, implementations MUST take several additional steps. As the exact behavior is mostly implementation specific, pseudocode will not be provided for most of the post processing steps.

1. Apply replay detection to the packet.

The `replay_tag` value returned by Sphinx_Unwrap MUST be unique across all packets processed with a given `private_routing_key`.

The exact specifics of how to detect replays is left up to the implementation, however any replays that are detected MUST be discarded immediately.

2. Act on the routing commands, if any.

The exact specifics of how implementations chose to apply routing commands is deliberately left unspecified, however in general:

- If there is a `NextNodeHopCommand`, the packet should be forwarded to the next node based on the `next_hop` field upon completion of the post processing.

  The lack of a NextNodeHopCommand indicates that the packet is destined for the current node.

- If there is a `SURBReplyCommand`, the packet should be treated as a SURBReply destined for the current node, and decrypted accordingly (See `Section 7.2`)

- If the implementation supports multiple recipients on a single node, the `RecipientCommand` command should be used to determine the correct recipient for the packet, and the payload delivered as appropriate.

  It is possible for both a RecipientCommand and a NextNodeHopCommand to be present simultaneously in the routing commands for a given hop. The behavior when this situation occurs is implementation defined.

3. Authenticate the packet if required.

If the packet is destined for the current node, the integrity of the payload MUST be authenticated.

The authentication is done as follows:

```
derived_tag = sphinx_packet.payload[:PAYLOAD_TAG_LENGTH]
expected_tag = ZEROBYTES( PAYLOAD_TAG_LENGTH )
if !CONSTANT_TIME_CMP( derived_tag, expected_tag ):
    /* Discard the packet with no further processing. */
    return ErrorInvalidPayload
```

Remove the authentication tag before presenting the payload to the application.

```
sphinx_packet.payload = sphinx_packet.payload[PAYLOAD_TAG_LENGTH:]
```

7. Single Use Reply Block (SURB) Creation

A Single Use Reply Block (SURB) is a delivery token with a short lifetime, that can be used by the recipient to reply to the initial sender.

SURBs allow for anonymous replies, when the recipient does not know the sender of the message. Usage of SURBs guarantees anonymity properties but also makes the reply messages indistinguishable from forward messages both to external adversaries as well as the mix nodes.

When a SURB is created, a matching reply block Decryption Token is created, which is used to decrypt the reply message that is produced and delivered via the SURB.

The Sphinx SURB wire encoding is implementation defined, but for the purposes of illustrating creation and use, the following will be used:

```
struct {
    SphinxHeader sphinx_header;
    opaque first_hop[NODE_ID_LENGTH];
    opaque payload_key[SPRP_KEY_LENGTH];
} SphinxSURB;
```

7.1 Create a Sphinx SURB and Decryption Token

Structurally a SURB consists of three parts, a pre-generated Sphinx Packet Header, a node identifier for the first hop to use when using the SURB to reply, and cryptographic keying material by which to encrypt the reply's payload. All elements must be securely transmitted to the recipient, perhaps as part of a forward Sphinx Packet's Payload, but the exact specifics on how to accomplish this is left up to the implementation.

When creating a SURB, the terminal routing_commands vector SHOULD include a SURBReply-Command, containing an identifier to ensure that the payload can be decrypted with the correct set of keys (Decryption Token). The routing command is left optional, as it is conceivable that implementations may chose to use trial decryption, and or limit the number of outstanding SURBs to solve this problem.

```
Sphinx_Create_SURB( additional_data, first_hop, path[] ) ->
                                                  sphinx_surb,
                                                  decryption_token
```

Inputs:

- `additional_data` The Additional Data that is visible to every node along the path in the header.
- `first_hop` The node id of the first hop the recipient must use when replying via the SURB.
- `path` The vector of PathHop structures in hop order, specifying the node id, public key, and routing commands for each hop.

Outputs:

- `sphinx_surb` The resulting Sphinx SURB.
- `decryption_token` The Decryption Token associated with the SURB.

The Sphinx_Create_SURB operation consists of the following steps:

1. Create the Sphinx Packet Header and SPRP key vector.

```
sphinx_header, payload_keys =
      Sphinx_Create_Header( additional_data, path )
```

2. Create a key for the final layer of encryption.

```
final_key = RNG( SPRP_KEY_LENGTH )
```

3. Build the SURB and Decryption Token.

```
SphinxSURB sphinx_surb;
```

```
sphinx_surb.sphinx_header = sphinx_header
sphinx_surb.first_hop = first_hop
sphinx_surb.payload_key = final_key

decryption_token = final_key + payload_keys /* Prepend */
```

7.2 Decrypt a Sphinx Reply Originating from a SURB

A Sphinx Reply packet that was generated using a SURB is externally indistinguishable from a forward Sphinx Packet as it traverses the network. However, the recipient of the reply has an additional decryption step, the packet starts off unencrypted, and accumulates layers of Sphinx Packet Payload decryption as it traverses the network.

Determining which decryption token to use when decrypting the SURB reply can be done via the SURBReplyCommand's id field, if one is included at the time of the SURB's creation.

```
Sphinx_Decrypt_SURB_Reply( decryption_token, payload ) -> message
```

Inputs:

- `decryption_token` The vector of keys allowing a client to decrypt the reply ciphertext payload. This decryption_token is generated when the SURB is created.
- `payload` The Sphinx Packet ciphertext payload.

Outputs:

- `error` Indicating a unsuccessful unwrap operation if applicable.
- `message` The plaintext message.

The Sphinx_Decrypt_SURB_Reply operation consists of the following steps:

1. Encrypt the message to reverse the decrypt operations the payload acquired as it traversed the network.

```
for i = LEN( decryption_token ) - 1; i > 0; --i:
    payload = SPRP_Encrypt( decryption_token[i], payload )
```

2. Decrypt and authenticate the message ciphertext.

```
message = SPRP_Decrypt( decryption_token[0], payload )

derived_tag = message[:PAYLOAD_TAG_LENGTH]
expected_tag = ZEROBYTES( PAYLOAD_TAG_LENGTH )
if !CONSTANT_TIME_CMP( derived_tag, expected_tag ):
    return ErrorInvalidPayload

message = message[PAYLOAD_TAG_LENGTH:]
```

8. Single Use Reply Block Replies

The process for using a SURB to reply anonymously is slightly different from the standard packet creation process, as the Sphinx Packet Header is already generated (as part of the SURB), and there is an additional layer of Sphinx Packet Payload encryption that must be performed.

```
Sphinx_Create_SURB_Reply( sphinx_surb, payload ) -> sphinx_packet
```

Inputs:

- `sphinx_surb` The SphinxSURB structure, decoded from the implementation defined wire encoding.
- `payload` The packet payload message plaintext.

The Sphinx_Create_SURB_Reply operation consists of the following steps:

1. Prepend the authentication tag, and append padding to the payload.

```
payload = ZERO_BYTES( PAYLOAD_TAG_LENGTH ) | payload
payload = payload | ZERO_BYTES( PAYLOAD_LENGTH - LEN( payload ) )
```

2. Encrypt the payload.

```
payload = SPRP_Encrypt( sphinx_surb.payload_key, payload )
```

3. Assemble the completed Sphinx Packet.

```
SphinxPacket sphinx_packet
sphinx_packet.header = sphinx_surb.sphinx_header
sphinx_packet.payload = payload
```

The completed `sphinx_packet` MUST be sent to the node specified via `sphinx_surb.node_id`, as the entire reply `sphinx_packet`'s header is pre-generated.

9. Anonymity Considerations

9.1 Optional Non-constant Length Sphinx Packet Header Padding

Depending on the mix topology, there is no hard requirement that the per-hop routing info is padded to one fixed constant length.

For example, assuming a layered topology (referred to as stratified topology in the literature) MIX-TOPO10, where the layer of any given mix node is public information, as long as the following two invariants are maintained, there is no additional information available to an adversary:

1. All packets entering any given mix node in a certain layer are uniform in length.
2. All packets leaving any given mix node in a certain layer are uniform in length.

The only information available to an external or internal observer is the layer of any given mix node (via the packet length), which is information they are assumed to have by default in such a design.

9.2 Additional Data Field Considerations

The Sphinx Packet Construct is crafted such that any given packet is bitwise unlinkable after a Sphinx_Unwrap operation, provided that the optional Additional Data (AD) facility is not used. This property ensures that external passive adversaries are unable to track a packet based on content as it traverses the network. As the on-the-wire AD field is static through the lifetime of a packet (ie: left unaltered by the `Sphinx_Unwrap` operation), implementations and applications that wish to use this facility MUST NOT transmit AD that can be used to distinctly identify individual packets.

9.3 Forward Secrecy Considerations

Each node acting as a mix MUST regenerate their asymmetric key pair relatively frequently. Upon key rotation the old private key MUST be securely destroyed. As each layer of a Sphinx Packet is encrypted via key material derived from the output of an ephemeral/static Diffie-Hellman key exchange, without the rotation, the construct does not provide Perfect Forward Secrecy. Implementations SHOULD implement defense-in-depth mitigations, for example by using strongly forward-secure link protocols to convey Sphinx Packets between nodes.

This frequent mix routing key rotation can limit SURB usage by directly reducing the lifetime of SURBs. In order to have a strong Forward Secrecy property while maintaining a higher SURB lifetime, designs such as forward secure mixes SFMIX03 could be used.

9.4 Compulsion Threat Considerations

Reply Blocks (SURBs), forward and reply Sphinx packets are all vulnerable to the compulsion threat, if they are captured by an adversary. The adversary can request iterative decryptions or keys from a series of honest mixes in order to perform a deanonymizing trace of the destination.

While a general solution to this class of attacks is beyond the scope of this document, applications that seek to mitigate or resist compulsion threats could implement the defenses proposed in COMPULS05 via a series of routing command extensions.

9.5 SURB Usage Considerations for Volunteer Operated Mix Networks

Given a hypothetical scenario where Alice and Bob both wish to keep their location on the mix network hidden from the other, and Alice has somehow received a SURB from Bob, Alice MUST not utilize the SURB directly because in the volunteer operated mix network the first hop specified by the SURB could be operated by Bob for the purpose of deanonymizing Alice.

This problem could be solved via the incorporation of a “cross-over point” such as that described in MIXMINION, for example by having Alice delegating the transmission of a SURB Reply to a randomly selected crossover point in the mix network, so that if the first hop in the SURB’s return path is a malicious mix, the only information gained is the identity of the cross-over point.

10. Security Considerations

10.1 Sphinx Payload Encryption Considerations

The payload encryption’s use of a fragile (non-malleable) SPRP is deliberate and implementations SHOULD NOT substitute it with a primitive that does not provide such a property (such as a stream cipher based PRF). In particular there is a class of correlation attacks (tagging attacks) targeting anonymity systems that involve modification to the ciphertext that are mitigated if alterations to the ciphertext result in unpredictable corruption of the plaintext (avalanche effect).

Additionally, as the PAYLOAD_TAG_LENGTH based tag-then-encrypt payload integrity authentication mechanism is predicated on the use of a non-malleable SPRP, implementations that substitute a different primitive MUST authenticate the payload using a different mechanism.

Alternatively, extending the MAC contained in the Sphinx Packet Header to cover the Sphinx Packet Payload will both defend against tagging attacks and authenticate payload integrity. However, such an extension does not work with the SURB construct presented in this specification, unless the SURB is only used to transmit payload that is known to the creator of the SURB.

Appendix A. References

Appendix A.1 Normative References

Appendix A.2 Informative References

Appendix B. Citing This Document

Appendix B.1 Bibtex Entry

Note that the following bibtex entry is in the IEEEtran bibtex style as described in a document called “How to Use the IEEEtran BIBTEX Style”.

```
@online{SphinxSpec,
title = {Sphinx Mix Network Cryptographic Packet Format Specification},
author = {Yawning Angel and George Danezis and Claudia Diaz and Ania Piotrowska an
url = {https://github.com/katzenpost/katzenpost/blob/master/docs/specs/sphinx.rst}
year = {2017}
}
```

COMPULS05

Danezis, G., Clulow, J., “Compulsion Resistant Anonymous Communications”, Proceedings of Information Hiding Workshop, June 2005, https://www.freehaven.net/anonbib/cache/ih05-danezisclulow.pdf

MIXMINION

Danezis, G., Dingledine, R., Mathewson, N., “Mixminion: Design of a Type III Anonymous Remailer Protocol”, https://www.mixminion.net/minion-design.pdf

MIXTOPO10

Diaz, C., Murdoch, S., Troncoso, C., “Impact of Network Topology on Anonymity and Overhead in Low-Latency Anonymity Networks”, PETS, July 2010, https://www.esat.kuleuven.be/cosic/publications/article-1230.pdf

RFC2119

Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels”, BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, http://www.rfc-editor.org/info/rfc2119

RFC5246

Dierks, T. and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2”, RFC 5246, DOI 10.17487/RFC5246, August 2008, http://www.rfc-editor.org/info/rfc5246

SFMIX03

Danezis, G., “Forward Secure Mixes”, Proceedings of 7th Nordic Workshop on Secure IT Systems, 2002, https://www.freehaven.net/anonbib/cache/Dan:SFMix03.pdf

SPHINX09

Danezis, G., Goldberg, I., “Sphinx: A Compact and Provably Secure Mix Format”, DOI 10.1109/SP.2009.15, May 2009, https://cypherpunks.ca/~iang/pubs/Sphinx_Oakland09.pdf