Post Quantum Sphinx

David Stainton

December 25, 2023

1 Abstract

This paper presents 'KEM Sphinx', an enhanced version of the Sphinx packet format, designed to improve performance through modifications that increase packet header size. Unlike its predecessor, KEM Sphinx addresses performance limitations inherent in the original design, offering a solution that doubles processing speed. Our analysis extends to the adaptation of KEM Sphinx in a post-quantum cryptographic context, showing a transition with minimal performance degradation. The study concludes that the trade-off between increased size and improved speed and security is justifiable, especially in scenarios demanding higher performance. These findings suggest KEM Sphinx as a promising direction for efficient, secure communication protocols in an increasingly post-quantum cryptographic landscape.

2 Introduction

The Sphinx packet format, as originally conceived, achieves remarkable compactness at the expense of performance. This paper proposes modifications to the Sphinx packet format, specifically designed to enhance performance at the trade-off of an increased packet header size. Our innovative construction, termed 'KEM Sphinx', significantly outperforms the traditional Sphinx construction, offering approximately double the speed. A major aspect of our analysis focuses on the transition from classical Sphinx, also known as NIKE Sphinx, to a hybrid post-quantum Sphinx. This transition demonstrates a negligible performance differential, a remarkable finding that contrasts sharply with the typical performance compromises observed in upgrading to hybrid post-quantum constructions.

3 NIKE Sphinx

The Sphinx paper [2] explains a clever group action blinding trick that they use to make the header very compact. Since we are using a NIKE, the Sphinx packet header contains a public key which the next hop uses to compute a DH

shared secret with it's private key. It again computes another group action for the blinding trick that computes the public key for the next hop. This let's us avoid having to stuff a bunch of public keys into the Sphinx packet header. Furthermore, with NIKE primitives such as X25519 which are very fast, these two public key operations are only 160233 nanoseconds on my old laptop.

3.1 PQ NIKE Sphinx with hybrid NIKE combiner

The NIKE cryptographic primitive itself can be a hybrid of classical and post quantum (e.g. X25519 and CTIDH[1]), with a simple combiner that appends the two shared secrets together. In the context of Sphinx this works fine since the two shared secrets are inputs to a KDF.

However besides CTIDH there doesn't seem to be any other choices of NIKE primitives that have any defense against sufficiently powerful post quantum adversaries. We'd like to use CTIDH in our hybrid Sphinx implementation but right now CTIDH implementations do not have good performance. And this poor performance is compounded by the NIKE Sphinx design requiring two public key operations.

4 KEM Sphinx Packet Structure

KEM Sphinx is a modification to the original Sphinx packet format where we use a KEM instead of a NIKE. This only requires one public key operation per hop instead of two.

Sphinx Geometry The Sphinx packet geometry is derived from the parameter constants and a few tunable parameters. Each Sphinx packet consists of two parts: the Sphinx Packet Header and the Sphinx Packet Payload. The packet header consists of several components which convey the information necessary to verify packet integrity and correctly process the packet, while the packet payload contains the application message data.

- $routing\_info\_len = per\_hop\_ri\_len * MAX\_HOPS$
- $header\_len = kem\_element\_len + routing\_info\_len + MAC\_LEN$
- $packet\_len = header\_len + payload\_len$

4.1 KEM Sphinx header

The elements of the KEM Sphinx packet are:

1. kem_ciphertext - KEM ciphertext encrypted to the next hop's public mix key

[1]CTIDH: Faster constant-time CSIDH (Commutative Supersingular Isogeny Diffie-Hellman)

2. route_info - 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.

3. mac - MAC, A message authentication code tag covering α and β.

4. payload - payload ciphertext

The header format has had two simple changes, the α field is now the KEM ciphertext rather than a NIKE public key (referred to as the group element in the Sphinx paper). And the per hop routing information ciphertext block now contains the KEM ciphertexts for the proceeding routing hops.

4.2 Create a KEM Sphinx Packet Header

Here we cover the KEM Sphinx packet header derivation in these next subsections. The followuing pseudo code examples are derived from our Sphinx specification document for Katzenpost. [1]

4.2.1 Derive key material for each hop

Here we derive the KEM ciphertext/shared secret and use the KDF to generate the three keys for each hop.

Algorithm 1 Derive key material for each hop

$num\_hops \leftarrow 3$ ▷ Let us assume 3 mix nodes.
$n \leftarrow 0$
$route\_keys \leftarrow []$
$kem\_elements \leftarrow []$
while $n \neq num\_hops$ **do**
    $sharedsecret_n, ciphertext_n \leftarrow ENCAPSULATE(mix\_public\_key_n)$
    $route\_keys[n] \leftarrow KDF(sharedsecret_n)$
    $kem\_elements[n] \leftarrow ciphertext_n$
    $n \leftarrow n + 1$
end while

4.2.2 Derive keystream and encrypted padding

Algorithm 2 Derive keystream and encrypted padding

$ri\_keystream \leftarrow []$
$ri\_padding \leftarrow []$
$n \leftarrow 0$
while $n \neq N$ **do**
    $ZERO \leftarrow ZERO(routing\_info\_len + per\_hop\_ri\_len)$
    $stream \leftarrow StreamCipher(route\_keys[n].header\_encryption)$
    $keystream \leftarrow XOR(ZERO, stream)$
    $ks\_len \leftarrow LEN(keystream) - ((n + 1) * per\_hop\_ri\_len)$
    $padding \leftarrow keystream[ks\_len :]$
    if $n > 0$ **then**
        $prev\_pad\_len \leftarrow LEN(ri\_padding[n])$
        $paddingB \leftarrow APPEND(ri\_padding[n - 1], padding[: prev\_pad\_len])$
        $padding \leftarrow XOR(padding[: prev\_pad\_len], paddingB)$
    end if
    $ri\_keystream \leftarrow APPEND(ri\_keystream, keystream[: ks_len])$
    $ri\_padding \leftarrow APPEND(ri\_padding, padding)$
end while

4.2.3 Create the routing info for the terminal hop

Algorithm 3 Create the routing info for the terminal hop

$i \leftarrow num\_hops - 1$
$zero\_blob \leftarrow ZERO(per\_hop\_ri\_len - LEN(path[i].routing\_commands))$
$ri\_fragment \leftarrow APPEND(path[i].routing\_commands, zero\_blob)$
$ri\_fragment \leftarrow XOR(ri\_fragment, ri\_keystream[i])$
$mac\_data \leftarrow APPEND(mac\_data, ri\_fragment)$
$mac\_data \leftarrow APPEND(mac\_data, ri\_padding[i-1])$
$mac \leftarrow MAC(oute\_keys[i].header\_mac, mac\_data)$
$routing\_info \leftarrow ri\_fragment$
if $num\_hops < MAX\_HOPS$ **then**
    $pad\_len \leftarrow (MAX\_HOPS - num\_hops) * per\_hop\_ri\_len$
    $routing\_info \leftarrow APPEND(routing\_info, RNG(pad\_len))$
end if

4.2.4 Calculate the routing info for the rest of the hops.

At the conclusion of this step, the following variables will be assigned values:

- routing_info - The completed routing_info block.
- mac - The MAC for the 0th hop.

Algorithm 4 Calculate the routing info for the rest of the hops.

```
1: i ← num_hops − 2
2: for i >= 0 do
3:     cmds_to_encode ← []
4:     j ← 0
5:     for j < LEN(path[i].routing_commands) do
6:         cmd ← path[i].routing_commands[j]
7:         if cmd.command == next_node_hop then
8:             cmd.MAC ← mac
9:         end if
10:        cmds_to_encode ← APPEND(cmds_to_encode, cmd)
11:        j ← j + 1
12:    end for
13:    ZERO ← ZERO(per_hop_ri_len − LEN(cmds_to_encode)
14:    ri_fragment ← APPEND(cmds_to_encode, ZERO)
15:    routing_info ← APPEND(ri_fragment, routing_info)
16:    routing_info ← XOR(routing_info, ri_keystream[i])
17:    data_to_mac ← []
18:    data_to_mac ← APPEND(data_to_mac, kem_elements[i])
19:    data_to_mac ← APPEND(data_to_mac, routing_info)
20:    if i > 0 then
21:        data_to_mac ← APPEND(data_to_mac, ri_padding[i − 1]
22:    end if
23:    mac ← MAC(route_keys[i].header_mac, data_to_mac)
24:    i ← i − 1
25: end for
26: return payload
```

4.2.5 Assemble the Sphinx packet header and payload keys vector

The Sphinx packet header is composed of:

- $kem\_element[0]$ from the above algorithm 1
- $routing\_info$ from the above algorithm 3
- mac from the above algorithm 3

Algorithm 5 Assemble the payload keys vector.

$payload\_keys \leftarrow []$
$i \leftarrow 0$
for $i < nr_hops$ **do**
    $payload\_keys \leftarrow APPEND(payload\_keys, route\_keys[i].payload\_encryption)$
    $i \leftarrow i + 1$
end for

At the conclusion of the header creation, the Sphinx header along with the vector of SPRP keys are returned to the user.

4.3 Create a Sphinx packet

Here we create the Sphinx Packet Header and SPRP key vector. Prepend the authentication tag, and append padding to the payload. Then we encrypt the payload and append it to the header and return the packet.

Algorithm 6 Create a Sphinx packet.

$sphinx\_header, payload\_keys \leftarrow sphinx\_create\_header(path)$
$payload \leftarrow APPEND(ZERO(payload\_tag\_len), payload)$
$payload \leftarrow APPEND(payload, ZERO(payload\_len - LEN(payload)))$
$i \leftarrow nr\_hops - 1$
for $i >= 0$ **do**
    $payload \leftarrow SPRP\_Encrypt(payload\_keys[i], payload)$
    $i \leftarrow i - 1$
end for
return $APPEND(sphinx\_header, payload)$

4.4 Unwrapping KEM Sphinx packets

Unwrapping KEM Sphinx packets is roughly twice as fast as the classical NIKE Sphinx because we removed one of the public key operations, we no longer calculate the group element for the next hop by blinding the current group element. Instead we extract the new KEM ciphertext from the encrypted routing information section of the Sphinx packet header.

Below we present a cryptographic circuit diagram of the KEM Sphinx unwrap operation with the following symbols used to signify the four elements of the Sphinx packet:

- α - kem_ciphertext
- β - route_info
- γ - mac
- δ - payload

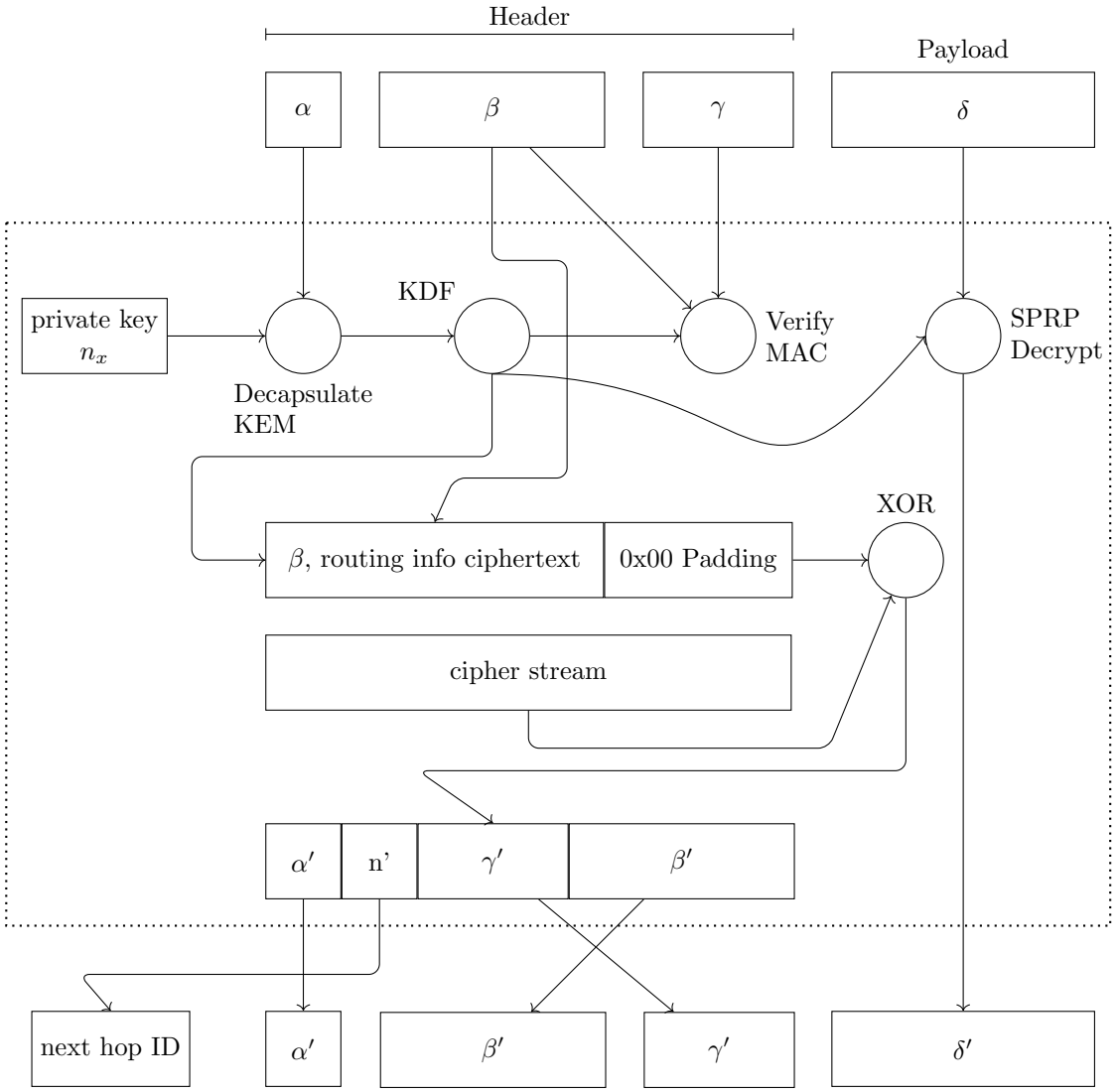

Figure 1: The processing of a KEM Sphinx message ((α, β, γ), δ) into ((α', β', γ'), δ') at Mix n.

We refer to this as the "unwrap" operation because it reveals either the plaintext payload or additional layers of encryption. The 'Sphinx_Unwrap' operation handles authentication, decryption and modifying the packet prior to forwarding it to the next node.

$Sphinx\_Unwrap(routing\_private\_key, sphinx\_packet) \rightarrow sphinx\_packet$,
$routing\_commands$,
$replay\_tag$,
$error$

The Sphinx unwrap operation takes two inputs:

1. Mix node's private key
2. A sphinx packet to unwrap

and produces four outputs:

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

The 'Sphinx_Unwrap' operation consists of the following steps:

4.4.1 Calculate the hop's shared secret

Algorithm 7 calculate the hop's shared secret

1: $hdr \leftarrow sphinx\_packet.header$
2: $shared\_secret \leftarrow DECAPSULATE(private\_routing\_key, hdr.kem\_element)$

4.4.2 Derive the various keys required for packet processing.

Algorithm 8 derive various keys

1: $keys \leftarrow Sphinx\_KDF(KDF\_INFO, shared\_secret)$

4.4.3 Validate the Sphinx Packet Header.

Algorithm 9 validate the Sphinx packet header

$to\_mac \leftarrow APPEND(hdr.kem\_element, hdr.routing\_info$
$derived\_mac \leftarrow MAC(keys.header\_mac, to\_mac))$
if $!CONSTANT\_TIME\_CMP(derived\_mac, hdr.MAC$ **then**
    return $erorrcode$
end if

4.4.4 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.

Algorithm 10 Extract the per-hop routing commands for the current hop.

$padding \leftarrow ZERO(PER\_HOP\_RI\_LENGTH)$
$B \leftarrow APPEND(hdr.routing\_info, padding)$
$cipher\_stream \leftarrow StreamCipher(keys.header\_encryption)$
$B \leftarrow XOR(B, cipher\_stream)$ ▷ Decrypt the entire routing_information block.

4.4.5 Parse the per-hop routing commands.

Algorithm 11 Parse the per-hop routing commands.

$cmd\_buf \leftarrow B[: PER\_HOP\_RI\_LENGTH - KEM\_CIPHERTEXT\_LENGTH]$
$new\_kem\_ciphertext \leftarrow B[PER\_HOP\_RI\_LENGTH - KEM\_CIPHERTEXT\_LENGTH : PER\_HOP\_RI\_LENGTH]$
$new\_routing\_info \leftarrow B[PER\_HOP\_RI\_LENGTH :]$
$next\_mix\_command\_idx \leftarrow -1$
$routing\_commands \leftarrow []$
$i \leftarrow 0$
for $i < PER\_HOP\_RI\_LENGTH$ **do**
    ▷ WARNING: Bounds checking omitted for brevity
    $cmd\_type \leftarrow B[i]$
    $cmd \leftarrow NULL$
    switch cmd_type **do**
        case NULL
            return ▷ No further commands.
        case next_node_hop
            $cmd \leftarrow RoutingCommand(B[i : i + 1 + LEN(NextNodeHopCommand)])$
            $next\_mix\_command\_idx \leftarrow i$ ▷ Save for later.
            $i \leftarrow i + 1 + LEN(NextNodeHopCommand)$
        case recipient
            $cmd \leftarrow RoutingCommand(B[idx : idx + 1 + LEN(FinalDestinationCommand)])$
            $i \leftarrow i + 1 + LEN(RecipientCommand)$
        case surb_reply
            $cmd \leftarrow RoutingCommand(B[i : i + 1 + LEN(SURBReplyCommand)])$
            $i \leftarrow i + 1 + LEN(SURBReplyCommand)$
    $i \leftarrow i + 1$
    $routing\_commands \leftarrow APPEND(routing\_commands, cmd)$ Append cmd to the tail of the list. */
end for

At the conclusion of the parsing step:

- 'new_kem_ciphertext' - The KEM ciphertext for the next hop.
- '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.

4.4.6 Decrypt the Sphinx packet payload

Algorithm 12 Decrypt the Sphinx packet payload

$payload \leftarrow sphinx\_packet.payload$
$payload \leftarrow SPRP\_Decrypt(key.payload\_encryption, payload)$
$sphinx\_packet.payload \leftarrow payload$

4.4.7 Transform the packet for forwarding to the next mix

Here we transform the packet for forwarding to the next mix, if the routing commands vector included a NextNodeHopCommand.

Algorithm 13 Transform the packet for forwarding to the next mix

if $next\_mix\_command\_idx != -1$ **then**
    $cmd \leftarrow routing\_commands[next\_mix\_command\_idx]$
    $hdr.kem\_element \leftarrow new\_kem\_ciphertext$
    $hdr.routing\_information \leftarrow new\_routing\_info$
    $hdr.mac \leftarrow cmd.MAC$
    $sphinx\_packet.hdr \leftarrow hdr$
end if

4.5 Sphinx packet post 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.

4.5.1 Apply replay detection to the packet.

Replay detection is accomplished by matching the hash of the shared secret which the node obtains when they DECAPSULATE the KEM ciphertext. We'll refer to this hash value as the replay tag. It must be unique across all packets processed with a given private key.

4.5.2 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
- 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.

4.5.3 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:

Algorithm 14 Authenticate packet

1: $derived\_tag \leftarrow sphinx\_packet.payload[: PAYLOAD\_TAG\_LENGTH]$
2: $expected\_tag \leftarrow ZEROS(PAYLOAD\_TAG\_LENGTH)$
3: **if** !CONSTANT_TIME_CMP(derived_tag, expected_tag) **then**
4: ▷ Discard packet.
5: **end if**

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

Algorithm 15 Remove authentication tag from payload

1: $sphinx\_packet.payload \leftarrow sphinx\_packet.payload[PAYLOAD\_TAG\_LENGTH :]$

5 Constructing PQ Hybrid KEMs

Special care must be taken in order correctly compose a hybrid post quantum KEM that is IND-CCA2 robust in the QROM. Most post quantum KEMs are IND-CCA2 however we must specifically take care to make our NIKE to KEM adapter have semantic security and we must make a security preserving KEM combiner.

The KEM Combiners paper [3] makes the observation that if a KEM combiner is not security preserving then the resulting hybrid KEM will not have IND-CCA2 security if one of the composing KEMs does not have IND-CCA2 security. Likewise the paper points out that when using a security preserving KEM combiner, if only one of the composing KEMs has IND-CCA2 security then the resulting hybrid KEM will have IND-CCA2 security.

Our KEM combiner uses the split PRF design from the paper.

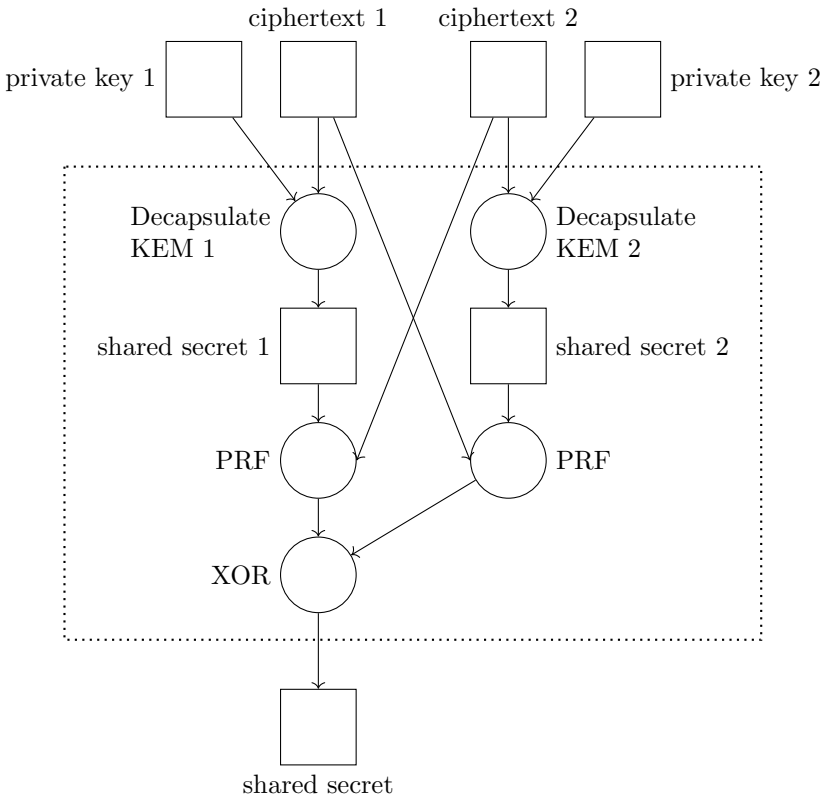

Figure 2: security preserving KEM combiner

Algorithm 16 Split PRF

function SPLIT_PRF($ss1$, $ss2$, $cct1$, $cct2$)
return $XOR(PRF(APPEND(ss1, cct2)), PRF(APPEND(ss2, cct1)))$
end function

If three or more KEMs are combined then the Split PRF is constructed like this:

Algorithm 17 Split PRF

function SPLIT_PRF($ss1$, $ss2$, $ss3$, $cct1$, $cct2$, $cct3$)
    $cct \leftarrow APPEND(cct1, cct2)$
    $cct \leftarrow APPEND(cct, cct3)$
    $ret \leftarrow XOR(PRF(APPEND(ss1, cct)), PRF(APPEND(ss2, cct)))$
    return $XOR(ret, PRF(APPEND(ss3, cct)))$
end function

5.1 NIKE to KEM Adapter

Our NIKE to KEM adapter is an ad hoc hashed ElGamal construction:

Algorithm 18 ad hoc hashed ElGamal construction

function ENCAPSULATE($their\_publickey$)
    $my\_privkey, my\_pubkey \leftarrow GEN\_KEYPAIR(RNG)$
    $ss \leftarrow DH(my\_privkey, their\_pubkey)$
    $to\_hash \leftarrow APPEND(ss, their\_pubkey)$
    $ss3 \leftarrow PRF(APPEND(to\_hash, my\_pubkey))$
    return $my\_pubkey, ss2$
end function
function DECAPSULATE($my\_privkey, their\_publickey$)
    $s \leftarrow DH(my\_privkey, their\_publickey)$
    $to\_hash \leftarrow appned(ss, my\_pubkey)$
    $shared\_key \leftarrow PRF(APPEND(to\_hash, their\_pubkey))$
    return $shared\_key$
end function

6 Security Evaluation

Proof of equivalence to the classical Sphinx goes here.

7 Implementations

7.1 Go implementation

8 Conclusion

KEM Sphinx generates cryptographic objects that are larger in size. However, depending on the specific application, the advantages in performance and security may justify the increased bandwidth overhead.

9 Acknowledgements

Special thanks to Peter Schwabe for encouraging me to look more closely into making KEM Sphinx. Also thanks to Bas Westerbaan and Bertram Poettering for answering my questions about KEM combiners.

10 Appendix I

All our pseudo code examples contain terms listed here:

- $GEN\_KEYPAIR(ikm) \rightarrow privkey, pubkey$
  Generate the KEM keypair given the initial key material.
- $ENCAPSULATE(public\_key) \rightarrow shared\_secret, kem\_ciphertext$
  Encapsulates to the given public key.
- $DECAPSULATE(my\_privkey, kem\_ciphertext) \rightarrow shared\_secret$
  Decapsulates the KEM ciphertext with the given private key.
- $ZERO(length\_in\_bytes) \rightarrow blob$
  Returns a blob of all zero (0x00) bytes.
- $LEN(byte\_slice) \rightarrow length$
  Returns the length of the byte slice.
- $KDF(ikm) \rightarrow keys$
  Generates three keys:
  1. header_mac
  2. header_encryption
  3. payload_encryption

- $XOR(blobA, blobB) \rightarrow output\_blob$
  Bitwise XOR operation.
- $APPEND(a, b) \rightarrow c$
  Returns b appended to a
- $ZERO(byte\_slice)$
  Overwrites the given byte slice with ZERO (0x00).
- $RNG(length) \rightarrow blob$
  Cryptographically secure random number generator.
- $MAC(key, data) \rightarrow mac$
  Produces a MAC value over the given data for the given key.
- $SPRP\_Encrypt(key, plaintext) \rightarrow ciphertext$
  Wide block cipher encrypt function given the key and plaintext.
- $SPRP\_Decrypt(key, ciphertext) \rightarrow plaintext$
  Wide block cipher decrypt function given the key and ciphertext.
- $PRF(data) \rightarrow hash$
  A cryptographic hash function.
- $DH(private\_key, public\_key) \rightarrow shared\_secret$
  Computes a diffiehellman shared secret.
- $StreamCipher(key) \rightarrow bit\_stream$
  Stream cipher.

11 Appendix II

11.1 KEM Sphinx Verifpal Proof

11.2 KEM Sphinx Proverif Proof

11.3 KEM Sphinx Tamarin Proof

[2]

References

[1] Yawning Angel et al. *Sphinx Mix Network Cryptographic Packet Format Specification*. 2017. URL: `https://github.com/katzenpost/katzenpost/blob/master/docs/specs/sphinx.rst`.

[2] George Danezis and Ian Goldberg. "Sphinx: A Compact and Provably Secure Mix Format". In: *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P 2009)*. Oakland, California, USA: IEEE Computer Society, May 2009, pp. 269–282. ISBN: 978-0-7695-3633-0.

[3] Federico Giacon, Felix Heuer, and Bertram Poettering. *KEM Combiners*. Cryptology ePrint Archive, Paper 2018/024. https://eprint.iacr.org/2018/024. 2018. URL: https://eprint.iacr.org/2018/024.