KEMSphinx

David Stainton

Abstract

Here I present a modification of the Sphinx cryptographic packet format that uses a KEM instead of a NIKE whilst preserving the properties of bitwise unlinkability, constant packet size and route length hiding.

Table of Contents

1. Introduction

We'll express our KEM Sphinx header in pseudo code. The Sphinx body will be exactly the same as the section called "References" [4] Our basic KEM API has three functions:

- `PRIV_KEY, PUB_KEY = GEN_KEYPAIR(RNG)`
- `ct, ss = ENCAP(PUB_KEY)` - Encapsulate generates a shared secret, ss, for the public key and encapsulates it into a ciphertext.
- `ss = DECAP(PRIV_KEY, ct)` - Decapsulate computes the shared key, ss, encapsulated in the ciphertext, ct, for the private key.

Additional notation includes:

- `||` = concatenate two binary blobs together
- `PRF` = pseudo random function, a cryptographic hash function, e.g. `Blake2b`.

Therefore we must embed these KEM ciphertexts in the KEMSphinx header, one KEM ciphertext per mix hop.

2. Post Quantum Hybrid KEM

Special care must be taken in order correctly compose a hybrid post quantum KEM that is IND-CCA2 robust.

The hybrid post quantum KEMs found in Cloudflare's circl library are suitable to be used with Noise or TLS but not with KEM Sphinx because they are not IND-CCA2 robust. Noise and TLS achieve IND-CCA2 security by mixing in the public keys and ciphertexts into the hash object and therefore do not require an IND-CCA2 KEM.

Firstly, our post quantum KEM is IND-CCA2 however we must specifically take care to make our NIKE to KEM adapter have semantic security. Secondly, we must make a security preserving KEM combiner.

2.1 NIKE to KEM adapter

We easily achieve our IND-CCA2 security by means of hashing together the DH shared secret along with both of the public keys:

```
func ENCAPSULATE(their_pubkey publickey) ([]byte, []byte) {
my_privkey, my_pubkey = GEN_KEYPAIR(RNG)
ss = DH(my_privkey, their_pubkey)
ss2 = PRF(ss || their_pubkey || my_pubkey)
return my_pubkey, ss2
}

func DECAPSULATE(my_privkey, their_pubkey) []byte {
s = DH(my_privkey, their_pubkey)
shared_key = PRF(ss || my_pubkey || their_pubkey)
return shared_key
}
```

2.2 KEM Combiner

The KEM Combiners paper ??? 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 when combining two KEM shared secrets together we use a hash function to also mix in the values of both KEM ciphertexts. In this pseudo code example we are hashing together the two shared secrets from the two underlying KEMs, ss1 and ss2. Additionally the two ciphertexts from the underlying KEMs, cct1 and cct2, are also hashed together:

```
func SplitPRF(ss1, ss2, cct1, cct2 []byte) []byte {
cct := cct1 || cct2
return PRF(ss1 || cct) XOR PRF(ss2 || cct)
}
```

Which simplifies to:

```
SplitPRF := PRF(ss1 || cct2) XOR PRF(ss2 || cct1)
```

The Split PRF can be used to combine an arbitrary number of KEMs. Here's what it looks like with three KEMs:

```
func SplitPRF(ss1, ss2, ss3, cct1, cct2, cct3 []byte) []byte {
cct := cct1 || cct2 || cct3
return PRF(ss1 || cct) XOR PRF(ss2 || cct) XOR PRF(ss3 || cct)
}
```

3. KEMSphinx Header Design

NIKE Sphinx header elements:

1. Version number (MACed but not encrypted)
2. Group element
3. Encrypted per routing commands
4. MAC for this hop (authenticates header fields 1 thru 4)

KEM Sphinx header elements:

1. Version number (MACed but not encrypted)

2. One KEM ciphertext for use with the next hop
3. Encrypted per routing commands AND KEM ciphtertexts, one for each additional hop
4. MAC for this hop (authenticates header fields 1 thru 4)

We can say that KEMSphinx differs from NIKE Sphinx by replacing the header's group element (e.g. an X25519 public key) field with the KEM ciphertext. Subsequent KEM ciphertexts for each hop are stored inside the Sphinx header "routing information" section.

First we must have a data type to express a mix hop, and we can use lists of these hops to express a route:

```
type PathHop struct {
public_key kem.PublicKey
routing_commands Commands
}
```

Here's how we construct a KEMSphinx packet header where path is a list of PathHop, and indicates the route through the network:

1. Derive the KEM ciphertexts for each hop.

```
route_keys = []
route_kems = []
for i := 0; i < num_hops; i++ {
kem_ct, ss := ENCAP(path[i].public_key)
route_kems += kem_ct
route_keys += ss
}
```

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

Same as in the section called "References" [4] except for the fact that each routing info slot is now increased by the size of the KEM ciphertext.

3. Create the routing_information block.

Here we modify the Sphinx implementation to pack the next KEM ciphertext into each routing information block. Each of these blocks is decrypted for each mix mix hop which will decrypt the KEM ciphertext for the next hop in the route.

4. Assemble the completed Sphinx Packet Header and Sphinx Packet Payload SPRP key vector. Same as in SPHINXSPEC except the `kem_element` field is set to the first KEM ciphertext, `route_kems[0]`:

```
var sphinx_header SphinxHeader
sphinx_header.additional_data = version
sphinx_header.kem_element = route_kems[0]
sphinx_header.routing_info = routing_info
sphinx_header.mac = mac
```

4. KEMSphinx Unwrap Operation

Most of the design here will be exactly the same as in SPHINXSPEC. However there are a few notable differences:

1. The shared secret is derived from the KEM ciphertext instead of a DH.
2. Next hop's KEM ciphertext stored in the encrypted routing information.

Acknowledgments

I would like to thank Peter Schwabe for the original idea of simply replacing the Sphinx NIKE with a KEM and for answering all my questions. I'd also like to thank Bas Westerbaan for answering questions.

References

KEMCOMB

Federico Giacon, Felix Heuer, Bertram Poettering, "KEM Combiners", 2018, https://link.springer.com/chapter/10.1007/978-3-319-76578-5_7

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

SPHINXSPEC

Angel, Y., Danezis, G., Diaz, C., Piotrowska, A., Stainton, D., "Sphinx Mix Network Cryptographic Packet Format Specification", July 2017, https://github.com/katzenpost/katzenpost/blob/main/docs/specs/sphinx.md