Ephemeral Messaging System

The Quantum Resistant Ledger

1 Introduction

The Quantum Resistant Ledger (QRL) is a fully quantum resistant blockchain network[6, 4], using hash-based signatures XMSS instead of ECDSA to secure transactions. The existing quantum security of the platform allows to extend the transaction capabilities to a whole suite of quantum-secure applications. In this paper, we describe the messaging protocol that we will build on top of the existing blockchain infrastructure: the Ephemeral Messaging System (EMS). EMS is a peer-to-peer broadcast messaging upgrade, where messages will be optionally relayed by QRL nodes that choose to route the traffic. It is a data messaging layer which is agnostic to the chain, but which extends the functionality of the network in new and interesting ways. Apps and services will be able to interact with EMS by means of a user friendly API, enabling asynchronous, authenticated end-to-end encrypted data communication between QRL nodes via the network.

These properties are achieved by utilizing quantum resistant public key encryption and stateless digital signatures. For maximal security, the quantum resistant algorithms will be combined with existing pre-quantum schemes, that have proven to be secure against normal computers for decades: a hybrid mode. To derive the keys for encryption and signatures, only the master seed of the wallet is used to derive the EMS key-pairs. Public keys are securely stored via XMSS on the QRL blockchain, by means of a "lattice transaction". By deploying this functionality, the QRL blockchain can also serve as a standalone public-key infrastructure by any third-party application. But the EMS protocol itself is able to secure messages back and forth, relayed through the QRL network. To prevent flooding of the capacity of the network, every message will have an expiry date, after which the network will drop the message. Protection against DDOS attacks is done via inclusion of a proof-of-work: the higher the work, the further the allowed expiry date.

2 Properties and Functionalities

The Ephemeral Messaging System will have several properties as standard supported, as well as additional privacy-extending properties and functionalities that will be build in at a later stage (See Section 5). These properties follow from the fact that we use specific building blocks (See Section 3) in the protocol, and are the following:

- **Hybrid post-quantum secure.** The cryptography in EMS will be hybrid post-quantum secure: post-quantum schemes coupled with normal cryptography. As NIST is still evaluating the schemes we want to use in EMS [5], we couple it with cryptography that has proven to be secure for many years. In the unlikely event that initial picked post-quantum schemes turn out insecure, EMS will still give full protection to anyone without quantum computers. This gives us time to transition to new quantum resistant schemes. Again, this is very unlikely but we want to give maximal protection to users.

- **Confidential and authenticated.** This means participants of a EMS chat know who they are talking to and are the only ones that can read the messages being sent back and forth. The confidentiality of the protocol stems from using hybrid encryption modes, where several extensions are possible to achieve more privacy for users (see Section 5). We will use Kyber [1] and elliptic-curves for the hybrid encryption. Authentication is taken care off by hybrid stateless digital signatures and by utilizing authenticated symmetric encryption algorithms for the data-channel. We will use Dilithium[3] and ECDSA for signatures and AES-GCM mode for authenticated symmetric encryption.

- **Ephemeral, or short-term liveliness.** EMS messages will be relayed by QRL nodes that are willing to do so. To avoid spamming of the nodes, every message will have an expiry

date. The expiry date can be a real-world time-stamp or a QRL blocknumber that lies in the future. Additionally, the EMS message will contain a proof-of-work value. The higher the work, the longer the message may live in the nodes. Depending on the network capacity, the proof-of-work can be adjusted higher or lower. If the message does not contain a valid proof-of-work, the message is dropped automatically by the network.

EMS will provide the following core functionalities as build-in. For more details on how this is implemented, see Section 4.

- **Public-key database on the QRL blockchain.** Using the so-called lattice transaction, users are able to upload their public keys to the QRL blockchain. The transaction contains both the two public keys for hybrid encryption, as well as the two public keys for hybrid signatures. The transaction is, like any transaction on the QRL blockchain, post-quantum secured via XMSS[2]. Although these transactions are bigger than regular QRL transactions, such a public key database opens the door to many exciting applications, where EMS chats is just one of them.
- **Off-chain ephemeral messaging via QRL nodes.** Together with lattice transactions that are stored on the QRL blockchain, EMS will also provide the possibility of sending messages via QRL nodes. These messages will not be stored on-chain, but rather off-chain on the QRL nodes that allow this storage. Each message has a specific format (See Section 4), that allows two parties to identify which messages belong to them. The protocol handles how encryption keys are used and how data is authenticated. Messages have a specific maximum size and expiration date. When users want to send EMS messages of bigger size, the message is chopped into multiple EMS messages and key-management is done via additional hashing functionalities (See Section 3).
- **External access via user-friendly API.** Apart from the off-chain ephemeral messaging capabilities described above, external applications can also access the data from EMS. For example, apps can use the QRL public-key database only and handle the encryption/signatures as they wish. EMS will provide some basic, secure functionalities such as public-key retrieval and signing and encryption of any message size to a specific QRL address with EMS keys on the QRL blockchain.

3 Building Blocks

To build the protocols we describe in the next Section, we first describe several building blocks in this section. The protocols are then nothing more then several steps using building blocks, as well as some sending/receiving data capabilities.

3.1 Hybrid public-key encryption scheme

The first thing we will introduce is the hybrid public-key encryption scheme. We will use a Key-Encapsulation-Mechanism (KEM in short), that will provide the symmetric key that we will use for data encryption. The hybrid-KEM, denoted by KEM^H, provides three algorithms: key-generation, encapsulation and decapsulation:

Key-generation $\text{KEM}^H_{\text{keygen}}(s) = (sk_{\text{K}}, pk_{\text{K}})$. Keys are generated using a seed s, which can be derived from the master-seed MS.

Encapsulation $\text{KEM}^H_{\text{enc}}(pk_{\text{K}}) = (C, K)$. A random, symmetric key K and corresponding cipher-text C is generated using the encapsulation algorithm with public-key pk as input.

Decapsulation $\text{KEM}^H_{\text{dec}}(sk_{\text{K}}, C) = K$. Using secret-key sk_{K} and cipher-text C, the symmetric key K can be derived.

3.2 Hybrid stateless digital signature scheme

We will add a hybrid stateless digital signature scheme to EMS, to ensure authentication and message integrity. It needs to be stateless, since EMS messages will be off-chain and asynchronous, meaning there will not be a way to store state about EMS messages. We denote the hybrid signature scheme by SIGN^H, and it provides the following three algorithms:

Key-generation $\text{SIGN}^H_{\text{keygen}}(s) = (sk_{\text{S}}, pk_{\text{S}})$. Keys are again generated using a seed s, which can be derived from the master-seed MS.

Signing $\text{SIGN}^H_{\text{sign}}(sk_{\text{S}}, m) = \sigma$. A signature σ is generated using the secret signing key sk_{S} and input message m.

Verification $\text{SIGN}^H_{\text{verify}}(pk_{\text{S}}, \sigma) = \{1, 0\}$. A signature σ is verified using public key pk_{S}, and is either success (1) or fail (0).

3.3 Authenticated Symmetric Encryption

Note that a KEM only provides to exchange a shared symmetric key K, so we need to use K in a symmetric encryption scheme to actually encrypt/decrypt messages in EMS. We will use an authenticated symmetric scheme to immediately authenticate data as well. There are two algorithms for this:

Symmetric encryption $\text{sym}_{\text{enc}}(K, M) = C$. On input of shared symmetric key K and message M, a cipher-text C is output. This cipher-text also contains data for authentication.

Symmetric decryption $\text{sym}_{\text{dec}}(K, C) = M$. On input of the shared symmetric key K and cipher-text C, a message M is output, together with a success or fail of authentication.

3.4 Hashing functionalities

We need several hashing functionalities, but all can be handled by using so-called eXtenable-Output-Functions (XoF). An example of a XoF is SHAKE, which is based on the SHA3 construction. Basically, it is a one-way random function, that maps any size input to random output of any desireable length. For example, when given the master-seed MS of a wallet, we want to transform this into seeds s for the KEM and the signature schemes. To do this, we can simply use SHAKE on input MS, together with a fixed string such as "KEM" or "SIGN" to get two different random seeds:

$$s_{\text{K}} = \text{SHAKE}(\text{“}KEM''||MS)$$
$$s_{\text{S}} = \text{SHAKE}(\text{“}SIGN''||MS)$$

Seeds s_{K} and s_{S} are random-looking to anybody and it is impossible, even for quantum computers, to retrieve master-seed MS from these values. This means that when a user logs in with their wallet (hardware or web), the seeds used for key-generation in the KEM and signature scheme can be derived, meaning that the user is able to decrypt EMS messages belonging to them or sign their own EMS message to send.

Another functionality is to expand a given, secret input to more random secret output bits. For example, when two users share a secret key K, they can expand it to any random output of any desirable size. Even if this whole output is given, it is again impossible to retrieve shared secret K from that. We can expand shared secret K into three different outputs Y_1, Y_2, Y_3, all of appropriate size, to be used later in the scheme:

$$Y_1, Y_2, Y_3 = \text{SHAKE}(K)$$

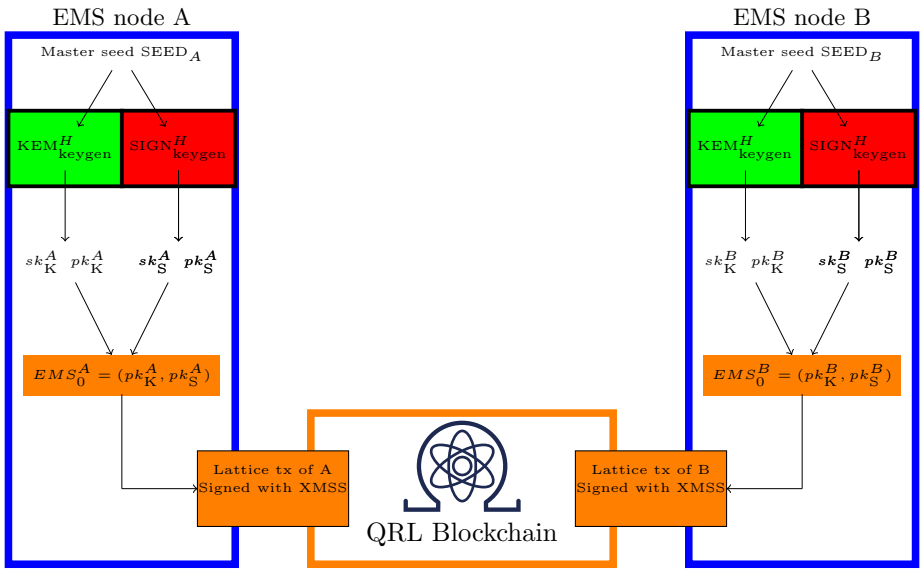
These outputs can again be used in SHAKE to generate even more output. And the nice thing is that these outputs are linked. If the other user has shared secret K as well, it can derive the same values Y_1, Y_2, Y_3 by using SHAKE and also any other outputs that are linked.

4 Protocol Descriptions

Now that we have defined the building blocks of EMS, the protocols are described simply by plugging in the building blocks in the right order.

4.1 Lattice-transaction: uploading the public keys

The first operation that is still on the chain of QRL is the lattice transaction. This transaction uploads the public keys for encryption (KEM^H) and the public keys for the stateless digital signatures (SIGN^H) on the QRL blockchain, using XMSS. A visualization of the protocol is given in the figure below.



As mentioned in Section 3, we will use SHAKE to transform a master-seed MS into new random seeds for key-generation in KEM^H and SIGN^H. The master-seeds are denoted in the figure by $\text{SEED}_A, \text{SEED}_B$ for two EMS nodes A and B. The key-generation outputs two key-pairs, one for the KEM and one for the digital signature scheme. The lattice transaction contains the public keys for the KEM pk_{K} and the public keys for the signature scheme pk_{S}. This is uploaded on the QRL blockchain and signed with XMSS. Two nodes A and B both need to do this to communicate via EMS. So in summary:

1. Use master-seed MS as input to shake, together with two different constant strings (for example "KEM" and "SIGN"), to generate two seeds s_{K} and s_{S}.
2. Generate keys using algorithms $\text{KEM}^H_{\text{keygen}}(s_{\text{K}})$ and $\text{SIGN}^H_{\text{keygen}}(s_{\text{S}})$, outputting KEM-keys $(sk_{\text{K}}, pk_{\text{K}})$ and signature keys $(sk_{\text{S}}, pk_{\text{S}})$
3. Initialize a lattice-transaction, containing public keys $(pk_{\text{K}}, pk_{\text{S}})$
4. Sign lattice-transaction with XMSS and upload to QRL blockchain

4.2 Off-chain EMS messaging

The message format EMS messages will have the following format:

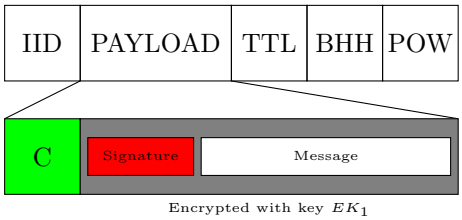
IID	PAYLOAD	TTL	BHH	POW

- IID: 64 byte identifier. It may be accessed as a complete ID field or two 32 byte separate identifiers (ID0 and ID1). The ID is intended to be a unique nonce
- PAYLOAD: max 8192 bytes body of the message containing user (encrypted) data
- TTL: 32 bit network expiry timestamp. After this expires, the message is dropped from the network
- BBH: 32 byte Block-Header Hash. Must be supplied as a rough message timestamp
- POW: 32 byte anti-spam proof-of-work nonce. The bigger the POW, the longer it may live in the network

The Initial Message The first message EMS_1 that one EMS node A sends to EMS node B contains the following message:

- IID: the first 32 bytes of the IID is the QRL wallet address of node B, denoted by Q^B. The second 32 bytes of the IID is the encryption of the sequence number (starting with $0\ldots01$ (32 bytes)) using key H_1. Key H_1 is generated from K_1 (described below) using SHAKE
- PAYLOAD: the payload contains several things
  1. Ciphertext C_1 generated by $\text{KEM}^H_{\text{enc}}(pk^B_{\text{K}}) := (C_1, K_1)$, where pk^B_{K} is node B's public key (that node B uploaded by a lattice transaction)
  2. Signature $\sigma_1 = \text{SIGN}^H_{\text{sign}}(sk^A_{\text{S}}, (C_1||Q^B))$ that signs C_1 and Bob's wallet address Q^B, using node A's signing key sk^A_{S}. The signature is encrypted together with the message.
  3. Encrypted message EM encrypted with key EK_1: $EM = \text{sym}_{\text{enc}}(EK_1, \sigma_1||M_1)$ where M_1 is the plaintext message. EK_1 is derived from K_1 using SHAKE.
- TTL: determined by POW and BBH
- BBH: last QRL block header hash during the sending of message
- POW: determines TTL

A schematic overview of EMS_1 is given below:

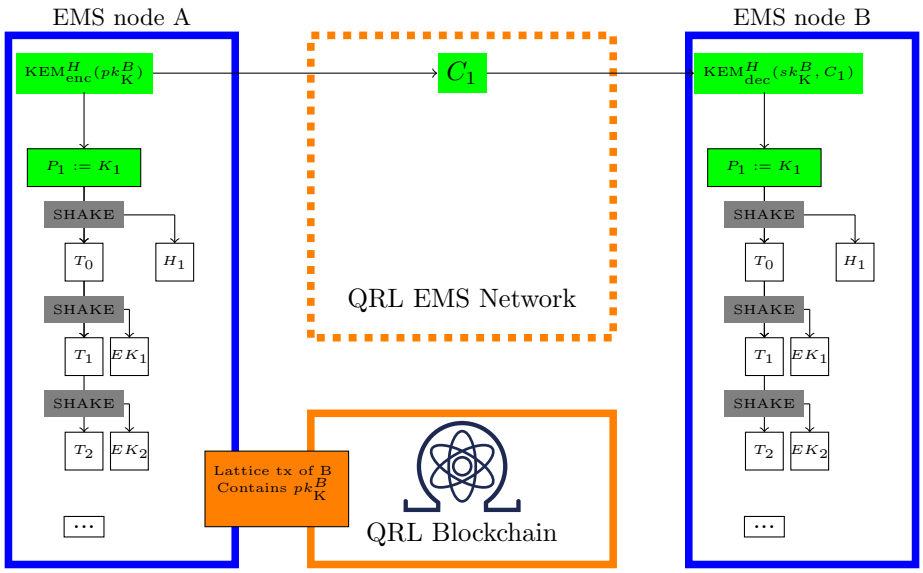


We derive the encryption keys (H_1 and EK_1) as follows:

1. From the KEM, we derive a shared secret key K_1 (node B can compute this via his secret key sk^B_{K} and ciphertext C_1.

2. This is used in SHAKEto determine T_0 and H_1: $T_0||H_1 := \text{SHAKE}(K_1)$
3. H_1 is the header encryption key used to derive shared IID for node A and node B
4. T_0 is used further with SHAKEagain to compute the real encryption keys: $T_1||EK_1 := \text{SHAKE}(T_0)$. Key EK_1 is used to encrypt the first plaintext message plus the signature σ
5. For subsequent messages, T_1 can be used to derive T_2 and EK_2, etc. (see further in this section)

Node B has to validate the signature σ using node A's public verification key, he can decrypt C_1 into K_1 and derive the exact same encryption keys H_1, EK_1 using SHAKE. A schematic overview is given below:



Subsequent Messages Subsequent messages (or should the first plaintext be too large for one EMS message) do not need a new ciphertext C using the public KEM, nor a new signature σ using SIGN: new keys can be derived using the original shared K_1 and SHAKE, if the first EMS message is still living in the network. The authentication is done using the symmetric authenticated encryption algorithm. To derive symmetric encryption keys $EK_2, EK_3, \ldots$, etc for EMS message $EMS_2, EMS_3, \ldots$, we only use SHAKEin the following sense:

$$\begin{aligned} T_1||EK_1 &:= \text{SHAKE}(T_0) \\ T_2||EK_2 &:= \text{SHAKE}(T_1) \\ T_3||EK_3 &:= \text{SHAKE}(T_2) \\ &\ldots \\ T_i||EK_i &:= \text{SHAKE}(T_{i-1}) \end{aligned}$$

The security of SHAKEensures the security of the encryption keys. The header encryption key H_1 does not need renewal.

Sending and receiving messages To send and receive message, both EMS users A and B will track 2 key-streams (with key-stream meaning EK_1, EK_2, etc): one for sending messages and one for receiving messages. This means that user B will redo the same operations as node A has done to send EMS_1.

4.3 External access via API

The fact that users can upload public keys to the QRL blockchain does not mean they are restricted in using EMS. Many functionalities will be provided by an API, such that external applications can use functionalities associated to the keys. Of course, to upload keys (the lattice transaction), the user needs a QRL address with quanta to actually make the transaction. Furthermore, there are several functionalities:

- **Encryption-only** Encryption can be done similar to how the payload is formed for EMS messages. The IID, TTL, BBH and POW are instead removed and only a ciphertext is output. This can be used to send encrypted messages to another user with a QRL address with public keys, via for example email or any other communication application.
- **Signature-only** Any message can be signed by a QRL user with public keys and anyone else (including users that do not have a QRL address) can verify the signature using the public verification key.
- **Sign-and-Encrypt** The functionalities of encryption and signatures can be combined in the sign-and-encrypt protocol. The message will be signed together with the QRL address of the receiver and will be encrypted. This payload can again be send over the internet using email or any other communication application.

5 Extensions

In this section we will discuss several extensions to the EMS procedure described so far, to improve privacy and security of the system.

5.1 Tunneling

Whilst it is possible to identify a channel opening message EMS_1 as it will include a receiver wallet address, it is possible to obfuscate which messages within a channel are linked by time. The order that messages are sent or received is not important – in fact the channel opening message could be the last message emitted to the network. Furthermore, it is possible place an EMS message EMS_v as the payload within another EMS message EMS_w. In this manner communication can be abstracted through an "EMS QRL gateway" and only relayed upon a given timestamp. Such measures would make identifying sender and receiver of a given series of ephemeral messages non-trivial.

5.2 Forward security

The fact that SHAKEcomputes encryption keys EK_1, EK_2 etc. in the forward direction, does not mean yet that the protocol is forward secure: this would mean that "stolen" encryption keys do not allow to decrypt messages that are send in the past. In order to achieve this property, the keys need to be dropped at the moment the key is used. For example, at the moment that T_0 and H_1 are computed via SHAKEfrom K_1, the key K_1 can be dropped by the user. Subsequently, when EK_1 is used by the user to encrypt message M_1, it may be dropped. Dropping these keys means the existence is no longer there, ensuring the forward security.

5.3 Re-keying

Unfortunately, there is still the case that "stolen" encryption keys allow to compute subsequent messages. For example, if K_1 is stolen (or worse, the secret keys belonging to the KEM), then the whole forward sequence generated by SHAKEcan be computed. To mitigate this, additional key material is needed to secure the connection in the backward direction: re-keying. After an

EMS node sends the first message EMS_1, he can furthermore ask the receiver to update the key-material. To do this, the sending EMS node A needs to produce new KEM keys: $(pk_{\mathrm{K}}^A, sk_{\mathrm{K}}^A)_2$ for example, where the subscript 2 denotes the second KEM key-material from node A. He then sends these keys to the receiving EMS node B. Node A does not need to do this for every message: that would not make sense. The only time EMS node A needs to compute $(pk_{\mathrm{K}}^A, sk_{\mathrm{K}}^A)_3$ (or 4,5,6,etc), is if he has received a message from node B after sending the previous new KEM keys (and only if they are used by node B). Node B can do similarly, which means that if keys get stolen, the updated keys from new KEM key-material can mitigate the risk of reading all future messages. Node B uses the new keys $(pk_{\mathrm{K}}^A, sk_{\mathrm{K}}^A)_2$ to generate a new ciphertext C_2 and key-material K_2, etc. To actually retrieve fresh encryption keys EK_2, EK_3, etc, node A (and node B) only need to update $P_1 := K_1$ in the following sense:

$$
\begin{aligned}
P_1 &:= K_1 \\
P_2 &:= \mathrm{SHAKE}(K_2 || P_1) \\
P_3 &:= \mathrm{SHAKE}(K_3 || P_2) \\
&... \\
P_i &:= \mathrm{SHAKE}(K_i || P_{i-1})
\end{aligned}
$$

Using P_i, both nodes can compute H_i and EK_i using SHAKE, as described in the protocol section. If this is done often, the keys are protected against break-ins.

References

[1] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - kyber: A cca-secure module-lattice-based KEM. *EuroS&P*, 2018.

[2] Johannes A. Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - A practical forward secure signature scheme based on minimal security assumptions. *PQCrypto*, 2011.

[3] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018.

[4] The QRL Foundation. The quantum resistant ledger. `https://theqrl.org/`.

[5] NIST. Post-quantum cryptography standardization. `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization`.

[6] Peter Waterland. Quantum resistant ledger (qrl). `https://github.com/theQRL/Whitepaper/blob/master/QRL\_whitepaper.pdf`.