# Tree authenticated ephemeral keys

Pavol Zajac

Slovak University of Technology in Bratislava, Slovakia
pavol.zajac@stuba.sk

**Abstract.** Public key algorithms based on QC-MPDC and QC-LDPC codes for key encapsulation/encryption submitted to NIST post-quantum competition (BIKE, QC-MDPC KEM, LEDA) are vulnerable against reaction attacks based on decoding failures. To protect algorithms, authors propose to limit the key usage, in the extreme (BIKE) to only use ephemeral public keys. In some authenticated protocols, we need to combine each key with a signature, which can lead to increased traffic overhead, especially given large signature sizes of some of the proposed post-quantum signature schemes. We propose to combine ephemeral public keys with a simple Merkle-tree to obtain a server authenticated key encapsulation/transport suitable for TLS-like handshake protocols.

**Keywords:** ephemeral keys, Merkle tree, authenticated KEM

## 1 Introduction

Recent NIST call for public-key post-quantum cryptographic algorithms [4] has motivated many researchers to propose new cryptographic schemes that are believed to be secure against quantum adversaries. One of the proposed candidates for Key Encapsulation is a suite of algorithms named BIKE by Aragon et.al. [1]. BIKE algorithms are based on quasi-cyclic binary moderate-density parity-check codes (QC-MDPC). A basic QC-MDPC KEM was also proposed by Yamada et.al. [13]. Another proposed system, LEDA by Baldi et.al., is based on QC low-density parity-check codes (QC-LDPC). The class of algorithms based on QC-LDPC and QC-MDPC codes can achieve post-quantum security with parameters similar to currently used RSA cryptosystem.

One of the main problems with LDPC/MPDC based designs is the probabilistic nature of decoding algorithms. Non-negligible decoding failure rate can lead to various attacks that can derive the secret key, such as [9, 6, 5]. To prevent these attacks, BIKE relies on generating one-time ephemeral keys. QC-MDPC KEM authors also recommend using ephemeral only keys. LEDA limits a number of uses for each key pair, but the number of key uses might be lower than originally proposed due to new attacks such as [7].

In TLS 1.3 handshake protocol [11], an initial session key is negotiated in key exchange using ephemeral public keys. The handshake is authenticated by signing protocol messages by the server, and verified by client using server's

certificate. Instead of (EC)DHE, a post-quantum public key cryptosystem or KEM with ephemeral keys, such as BIKE, QC-MDPC KEM or LEDA, can be used to transport initial key shares to provide post-quantum security. To fully migrate TLS handshake to the post-quantum era, we also need to replace server side signatures and certificates with quantum-resistant routines. The signature structure can be compatible with previous TLS 1.2 handshake with signed keys.

There are multiple candidates for post-quantum signatures with varying public key and signature sizes. In a typical handshake scenario, both a public key and a signature needs to be sent. Among balanced-size algorithms, Falcon signature [8] scheme (lattice based) provides Level-1 security with 897 bytes public key and 617 bytes (on average) signature size. Short signatures are achieved by multivariate proposals, led by HiMQ-3F [12] with 67 bytes needed for signature, and 100878 bytes for public key. On the opposite side, hash-based signature schemes provide short public keys and long signatures, led by Sphincs+ [3] with 32 byte public key and 8080 byte signature.

Hash-based signature schemes are based on one-time signatures (OTS) that are combined with Merkle trees or Goldreich trees to provide multiple-time signatures. Our observation is, that when we use ephemeral public keys, we already have a one-time key pair with its own secret. Thus, we do not need to produce a specific one-time signature, we only need to pre-authenticate the sequence of public keys. We propose to use Merkle trees with leaves directly based on hashes of ephemeral public key sequence. An $l$-level Merkle tree will allow us to authenticate $2^l$ ephemeral trees, with the public key size equal to the size of a single hash value, and another $l-1$ hash values needed for the signature (to store the path through the Merkle tree). E.g. an $l = 20$ level Merkle tree will provide authentication for $2^{20}$ ephemeral keys, with the total signature + PK size for 128-bit security level equal to 640 bytes. This comes at the additional cost of storing the hash tree on the server with $2^l$ hashes (32 MB in the previous setup).

If the number of one-time certificates is not sufficient, with PKI support, the root of the Merkle tree can be signed by the CA with an algorithm with different trade-off. As the public key of the CA can be pre-installed in the client device, CA's algorithm can have a short signature size and large public key. Different trade-offs with sizes of Merkle trees and certification paths can be made for different usage scenarios (depending on the number of accesses, available storage, tree precomputation time, etc.). E.g., we can construct another simple Merkle tree on top of $2^{20}$ OTS on the server as a level one local certification authority (whose root is further signed by real authority). This will provide a flexible server setup with $2^{40}$ usable keys.

The stateful authentication using Merkle tree based authentication of public keys retains the forward secrecy, because we can derive private keys on the fly from a seed using a one-way function (old private keys are not needed anymore after they are used). We can modify the scheme to stateless, if we relax conditions on a key reuse (e.g. for the use with LEDA cryptosystem [2]). In this case, however, we will lose the forward secrecy, because we must store the private

keys, or the pre-key, that can be used to derive any private key authenticated in the Merkle tree.

## 2 Preliminaries

We will work in the client-server scenario typical for internet communication. We rely on standard cryptographic tools and primitives:

- Public key cryptosystem with a pair of keys. We are only interested in key generation and authentication. $KeyGen$ primitive for a cryptosystem should efficiently generate a pair $(SK, PK)$ (from some randomness, see further), where $SK$ denotes a secret key, and $PK$ a public key. We suppose that to initiate secure communication between the client and server, it is sufficient to provide a mechanism to transport the authenticated public key of the server to the client. We are not interested in further protocols that realize the rest of the secure channel establishment, etc.
- $KeyGen$ primitive can be based on a deterministic algorithm $KDF : \mathbb{Z}_2^n \to \mathcal{K}$, that computes keypair $(SK, PK)$ from a bitstring $k$ of length $n$. We will call $k$ a pre-key. In classical setting $n = \lambda$, where $\lambda$ is a security level, but in post-quantum setting we will use pre-keys $n = 2\lambda$ to prevent Grover's algorithm based speed-up.
- A truly random pre-key is required for a secure public-key system. In our scheme we will use a single master (secret) pre-key that is generated as a true random bit-string. All other pre-keys will be derived with a one-way function $OWF : \mathbb{Z}_2^n \to \mathbb{Z}_2^n$.
- In the construction of public key authentication, we will also use a specific cryptographically secure hash function denoted by $hash$ (in practice instantiated by the standard SHA-2 or SHA-3). Both $OWF$ and $KDF$ can be implemented with a correct use of the same hash function (or by a different specific mechanism, as required by the system/protocol).

## 3 Merkle tree

Merkle tree was introduced in [10] to allow signing arbitrary documents with any one-way function $F$. It is a basis of post-quantum secure hash-based signatures. The Merkle tree is used in conjunction with one-time signatures (e.g. Lamport or Winternitz), and extends them for arbitrary number of uses.

We use a simplified hash tree method, based on binary tree with $l$ levels. Let $h_1$ denote the root of the tree, and let each node $h_i$ have a left sub-node $h_{2i}$ and a right sub-node $h_{2i+1}$. Each node is marked by a hash of subnodes $h_i = hash\,(h_{2i}|h_{2i+1})$.

Let us suppose that the root node $h_1$ is publicly known. It suffices to prove the knowledge of the whole hash tree by publishing a single hash path from some requested $h_j$. The path consists of the missing sub-node $h_{2i}$ or $h_{2i+1}$ required to compute the $h_i$, up to the previously published root $h_1$.

Note that we can restrict the essentially infinite Merkle tree to a fixed number of levels, say $l$. Then the final leaves have indexes between $2^l$ and $2^{l+1} - 1$ (here $l = 0$ meaning degenerate tree with only the root node).

## 4 Combining Merkle tree and ephemeral keys

Our goal is to provide authentication for the server's ephemeral key pairs. We can understand this as a problem of authenticating a set of public keys, each of which can be used only once. The corresponding secret keys are never revealed, but their ownership is verified in the course of the underlying protocol.

The set of private keys can be randomly generated, and stored in a key store. A more efficient option is to derived private keys from some master secret (a secret pre-key). In the pre-key approach, we can either derive consecutive secret keys from the pre-keys chained in a sequence (with a one-way function $OWF$ that derives the next pre-key from the previous). Alternatively, we can derive each secret key from a master pre-key, and an index in the set of keys. The sequential generation has the advantage of forward-secrecy: the server can store only the last pre-key. The price is, however, handling the issues with concurrency and state-keeping. The index method allows a (pseudo-)random selection of public keys (with no state and parallel access to the server). As the design goal of TLS 1.3 was to provide default forward secrecy, we only present the version with sequential pre-keys.

To generate an authenticated set of ephemeral keys, server does the following pre-computation:

1. Generate a random secret seed $k_0 \in \mathbb{Z}_2^\lambda$.
2. Use a one way function $F$ to define a sequence of derived pre-keys $k_i = OWF(k_{i-1})$.
3. Generate $2^l$ ephemeral key pairs $(SK_i, PK_i)$ from pre-keys $k_i$ using the defined $KDF$ function.
4. Compute the hashes $h_{2^l+i} = hash(PK_i)$, and the rest of the Merkle tree with $h_j = hash\,(h_{2j}|h_{2j+1})$.
5. Publish (sign by CA) the root $h_1$.
6. Store as an initial (secret) state: $(0, k_0)$ and the hash path from $h_{2^l}$ to the root.

To initiate the KEM, the server does:

1. Send current public key $PK_i$ along with hash path from $h_{2^l+i}$ to the root.

To verify the authenticity of the public key, the client does:

1. Verify that $h_{2^l+i} = hash(PK_i)$, and that for each hash in the path to the root: $h_j = hash\,(h_{2j}|h_{2j+1})$.

After the handshake part is finished, and keypair $(SK_i, PK_i)$ is no longer needed, server prepares a new key pair:

1. Derive the next $k_{i+1} = OWF(k_i)$.
2. Recompute the hash path.
3. Store a new (secret) state: $(i+1, k_{i+1})$ and the hash path from $h_{2^l+i+1}$ to the root.

To recompute the hash path, there are two possibilities. Either the whole hash tree needs to be stored (if enough storage is available), in which case generating a new hash path is trivial. We only need to store public key hashes, and do not need to store the key pairs or seeds $k_j$ with $j \neq i$. However, if key generation is too slow, we might prefer to store the key-pairs as well (but this option is more vulnerable to data-retention attacks). It is also possible to combine key derivation with more efficient batch key generation (such as the one for BIKE-2 in [1]).

We can save some storage space by recomputing the required parts of the Merkle tree after the key use. We only need to store the current $k_i$ and the path from $h_{2^l+i}$ to the root. All the left-hand sub nodes for the path from $h_{2^l+i+1}$ are actually computed when verifying the path from $h_{2^l+i}$. Only the right-hand child nodes need to be recomputed, similar to original hash tree precomputation. In this case, we never need to store $k_j$ with $j < i$, thus old seed values can still be removed (to preserve forward secrecy).

If we allow key reuse and do not need forward secrecy, we can change the computation of pre-keys to $k_i = hash(k_0|i)$. In this case, in KEM initiation phase, server selects $PK_i$ randomly among $2^l$ options. This factor limits a potential attacker's access to private keys by factor $2^l$.

Note that a man in the middle type attacker can always present a valid signed public key to the client in both cases: the signature of any of the previously used public keys is known. However, the attacker does not know the corresponding private key (and the signature does not help him to learn it). In the stateful version of the protocol, the past private keys can not be computed even by the server, which achieves forward secrecy. Attacker cannot forge the signature of any public key, that is not already stored in the Merkle tree due to the properties of the Merkle tree and the underlying hash function.

## 5 Basic security analysis

In this paper we have introduced a general scheme that combines public key encryption with Merkle trees for key authentication. Security of the scheme is based on its components and should be analysed in concrete instantiation. However, we can provide a simple security analysis for a general scheme with random key pairs.

Suppose that there is an $M$ level Merkle tree that is used to authenticate $2^M$ random public keys with (pre-shared) root $h_1$. We will only focus on authentication. Thus, we suppose that attacker has a knowledge of all public keys and the whole Merkle tree, but he is unable to learn any information about secret keys. Furthermore, we suppose some ideal hash function $h : \mathbb{Z}_2^* \to \mathbb{Z}_2^n$ was used to build the Merkle tree: $h$ is essentially a collision resistant one-way function.

The security arguments for the scheme are really simple. Attacker can present any public key $PK$ with any authentication path $(H_1, H_2, \ldots, H_{M-1})$. Attacker "wins" if the $PK$ gets accepted by verifier. This happens only if after evaluating hash path with the initial $h(PK)$ the verifier gets $H_M$.

It is easy to see that if the attacker provided a valid $H_M$ to the verifier, he was able to create a preimage of a hash function $h$ in some of the points of the original Merkle tree. For the sake of simplicity, suppose that authentication path is always evaluated from the right-side. Attacker's $PK$, and sequence $(H_1, H_2, \ldots, H_{M-1})$ induce a sequence of hash values $g_0 = h(PK)$, $g_i = h(g_{i-1}|H_i)$. Attacker's success means that $g_M = h_1$. This means that either attacker has constructed a preimage for $h_1$, or that $g_{M-1} = h_2$, and $H_{M-1} = h_3$, respectively. We can now repeat the argument to finding preimage of $h_2$, $h_4$, etc. up to $h_{2^M}$ (which is the hash of an original public key in authenticated by the Merkle tree).

We have thus established that if attacker succeeded, he has constructed a preimage for $h$. Thus, authentication property of the scheme reduces to preimage resistance of the used hash function. Note however, that there is a slightly more complicated situation: attacker can try to create preimage to any of the $2^{M+1}$ hash values included in the Merkle tree. In ideal case, probability of creating a preimage with random tries is $2^{M+1}/2^n$. Thus the attacker needs to test approximately $2^{n-M}$ values to succeed with at least 50% probability. Note that he does not need to generate as many public keys, as he can try to combine hashes of randomly generated public keys with hashes already in the public Merkle tree. However, this optimisation does not reduce the expressed work-factor expressed as number of hash calls.

If hash size $n = 2\lambda$, where $\lambda$ is a chosen security level, the Merkle tree can contain up to $M = \lambda$ levels to resist preimage attacks. In practice, $M$ is significantly lower, and the generic attack is not possible.

## 6 Conclusions

Public key encryption schemes / or KEMs / with ephemeral keys only (or keys with a limited number of uses) are sufficient to build TLS-like handshake protocols. While the authentication of the protocol can be solved with additional signature scheme, we can also authenticate the ephemeral public keys directly by employing Merkle-tree technique. In post-quantum setting this technique can provide a building block to more efficient and flexible authenticated key establishment schemes.

Selected components and their well known security properties should provide tools to create a key exchange mechanism (with underlying KEM) with server authentication (through the use of Merkle tree based signatures) and optionally forward secrecy (by the use of ephemeral keys). Authentication can be extended to the client side as well, if the client also keeps a Merkle tree of her potential ephemeral public keys. Client authenticated keys can also be recommended to prevent potential key exhaustion attack (malicious attacker creates false connections to a server to exhaust one-time keys). In the TLS setting, the lifetime of

the set of authenticated keys can also be extended by using pre-shared keys for clients that re-connect to the server, and with a suitable use of hierarchical PKI.

## References


1. Aragon, N., Barreto, P., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Gaborit, P., Gueron, S., Guneysu, T., Melchor, C.A., et al.: BIKE: Bit flipping key encapsulation (2017)
2. Baldi, M., Barenghi, A., Chiaraluce, F., Pelosi, G., Santini, P.: Design of ledakem and ledapkc instances with tight parameters and bounded decryption failure rate
3. Bernstein, D., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S.L., Hülsing, A., Kampanakis, P., Kölbl, S., Lange, T., Lauridsen, M., Mendel, F., R., N., Rechberger, C., Rijneveld, J., Schwabe, P.: SPHINCS+ (2017), https://sphincs.org/
4. Chen, L., Chen, L., Jordan, S., Liu, Y.K., Moody, D., Peralta, R., Perlner, R., Smith-Tone, D.: Report on post-quantum cryptography. US Department of Commerce, National Institute of Standards and Technology (2016)
5. Eaton, E., Lequesne, M., Parent, A., Sendrier, N.: Qc-mdpc: A timing attack and a cca2 kem. In: International Conference on Post-Quantum Cryptography. pp. 47–76. Springer (2018)
6. Fabšič, T., Hromada, V., Stankovski, P., Zajac, P., Guo, Q., Johansson, T.: A reaction attack on the QC-LDPC McEliece cryptosystem. In: International Workshop on Post-Quantum Cryptography. pp. 51–68. Springer (2017)
7. Fabšic, T., Hromada, V., Zajac, P.: A reaction attack on ledapkc. IACR eprint archive (2018), https://eprint.iacr.org/2018/140
8. Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon: Fast-fourier lattice-based compact signatures over ntru (2018)
9. Guo, Q., Johansson, T., Stankovski, P.: A key recovery attack on MDPC with CCA security using decoding errors. In: Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22. pp. 789–815. Springer (2016)
10. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) Advances in Cryptology — CRYPTO '87. pp. 369–378. Springer Berlin Heidelberg, Berlin, Heidelberg (1988)
11. Rescorla, E.: The transport layer security (tls) protocol version 1.3. Tech. rep. (2018)
12. Shim, K.A., Koo, N., Park., C.M.: HiMQ-3: A high speed signature scheme based on multivariate quadratic equations (2017)
13. Yamada, A., Eaton, E., Kalach, K., Lafrance, P., Parent, A.: QC-MDPC KEM: A key encapsulation mechanism based on the QC-MDPC McEliece encryption scheme (2017), https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions