AMS - Abstract Merkle Signature Scheme

Author: Herman Schoenfeld *<herman@sphere10.com>*
Version: 1.1
Date: 2020-07-20

Abstract

An abstract post-quantum digital signature scheme is presented that parameterizes a one-time signature scheme (OTS) for "many-time" use. This scheme permits a single key-pair to efficiently sign and verify a (great) many messages without security degradation. It achieves this by following the original Merkle-Signature Scheme but without a coupling to a specific OTS. Various improvements include a reduction in signature size, resistance to denial-of-service attacks and smaller keys. This construction comprises a bit-level specification for the Abstract Merkle Signature Scheme (AMS).

1. Introduction

Abstract Merkle Signatures (AMS) are a class of a quantum-resistant digital signature schemes that utilize hash-based cryptography without any dependency on elliptic-curves or discrete logarithms. AMS is a formalization of the scheme originally proposed by Ralph Merkle [1] but in an OTS-agnostic manner.

AMS is a "high-level abstract" scheme that takes as a parameter a one-time signature scheme (OTS) and transforms it into a "many-time" equivalent. To this end, AMS serves as an "algorithm wrapper" of the parameterized OTS algorithm. AMS achieves this goal by encapsulating the cryptography of the OTS in an ephemeral manner[^*] and by isolating the complexity of multiple OTS keys through the use of merkle-trees.

AMS is similar in goal and scope to XMSS [2] but without a coupling to a specific OTS, with simplified tree structuring and with DoS-vulnerability hardening. An AMS algorithm generally retains the performance, memory and security characteristics of the underlying OTS scheme but adds additional computational complexity in it's key generation. This arises from the fact that an AMS key is essentially a commitment to many pre-generated OTS keys.

In practice, an AMS implementation comprises of two layers, the OTS and AMS layers respectively. The AMS-layer of an AMS-algorithm requires the use of a cryptographic hash function (CHF) which is typically chosen to be the same as that employed by the OTS-layer of the algorithm (although this is not a strict requirement). The AMS-layer is itself quantum-resistant as it relies solely on the proper use of a cryptographic hash function. Thus, if the OTS-layer is quantum-resistant then the full AMS-algorithm is quantum-resistant.

Whilst this document focuses on the AMS-layer and the integration points that an OTS-layer must comply with, it does not cover specific implementations of OTS-layers.

[*]: *When an OTS algorithm is encapsulated within an AMS algorithm, the OTS key and signature objects are always short-lived and only computed when needed. In this sense they are ephemeral. As such, only the AMS key and signature objects are ever persisted or transmitted.*

2. AMS Scheme

AMS is a general-purpose, quantum-resistant cryptographic scheme offering the following features:

- Multi-use keys: a single private/public key-pair can be used to securely sign and verify many messages.
- Compact keys: keys are small and contain (compressed) hash commitments to OTS keys.
- Efficient comparisons: testing if a public key derives from a private key is time efficient.
- Parameterized: the underlying OTS and CHF can be changed without affecting the AMS-layer whatsoever.
- Quantum-resistant: AMS inherits all the PQC characteristics of the selected OTS algorithm.

Whilst AMS is strictly a "stateful" algorithm in that the signer must "remember information" about the most recent signature, it can used in a "practically stateless" manner in blockchain/DLT use-cases. This is possible since changes to the key-store are not required in this scheme, only remembering the index of last used OTS key is necessary. Thus in blockchain/DLT applications, rather than maintaining this index in a local key-store, the blockchain maintains a public strictly increasing nonce that is associated with the signer. Every time a signature is generated by the signer, the public nonce field is correspondingly incremented on the public ledger as part of the consensus rules. So long as this nonce is atomically and strictly increased within the same transaction that contains the signature, no risk key re-use arises (which could if relying on local data-stores susceptible to local corruption).

The AMS-layer of the scheme relies on a single cryptographic hash function (CHF) for all processing without any dependency on elliptic-curves or discrete logarithms. This CHF is also parameterized and, by convention, chosen to match that of the selected OTS scheme (although not strictly required).

The construction can be basically summarized as follows:

1. A **"Private Key"** is a user secret that deterministically generates batches of OTS key-pairs.
2. A **"Public Key"** is a merkle-root of a batch of OTS key-pairs.
3. A **"Signature"** comprises of an OTS signature, an OTS public key to validate that signature, and a merkle-proof of that key within a "batch".
4. Signature verification entails both the underlying OTS verification algorithm and a merkle-proof verification of the OTS key in the batch.

2.1 Notation & Definitions

1. `||` is operator that denotes byte array concatenation. If the operand is a `BYTE`, `DWORD`, `QWORD` it is implicitly converted to byte array using `ToBytes` function.
2. `H(x)` is a one-way cryptographic hash function (CHF) of `U`-bits.
3. `H^n` is cryptographic hash function that iterates the `H` function `n` times such that `H^0(x) = x, H^1(x) = H(x), H^2 = H(H(x)), etc`.
4. `LastDWord(arr)` is a function that extracts the last 4 bytes of byte array `arr` and re-interprets it as a little-endian 32-bit unsigned integer.
5. `LastQWord(arr)` is a function that extracts the last 8 bytes of byte array `arr` and re-interprets it as a little-endian 64-bit unsigned integer.

6. `RandomBytes(N)` is a function that returns an array of `N` cryptographically random bytes.
7. `ToBytes(N)` is a function takes an unsigned 64-bit (or 32-bit) integer argument `N` returns it was an array of 8 (or 4) bytes in little-endian layout.
8. Unless otherwise specified, all byte layouts are in little-endian format by default.

2.2 Private Key

A Private Key `P` is generated as follows:

`P = v || OTS || h || RESERVED || Entropy`

Field	**Description**	**Bits**	**Value Range**
`v`	AMS version	8	1..256
`OTS`	OTS algorithm	16	1..65536
`h`	Height parameter	8	0..255
RESERVED	30 bytes reserved for OTS parameters	224	0
Entropy	Cryptographically random bytes used to seed OTS key generation	256	CRNG

A private key is `64` bytes in length and can be used to sign up to `2^(64 + h)` messages. A Private Key can derive up to `2^64` unique Public Keys.

2.2.1 Field: Version

Version is an 8-bit value mapping to integers `1..256`. As of this revision, the version is always `1` (mapping to all zeros). Values `2..256` are reserved for future revisions of the AMS scheme which can evolve independently from underlying OTS schemes.

2.2.2 Field: OTS

The OTS field is a 16-bit field mapping to the integers `1..65536` which determines the AMS algorithm being used. These values are globally allocated by the author. Currently, they are:

AMS Algorithm	**Description**	**OTS Algorithm**	**Value**
LAMS	Lamport Abstracted Merkle Signatures	Lamport	1
WAMS	Winternitz Abstracted Merkle Signatures	W-OTS	2
WAMS+	Winternitz Abstracted Merkle Signatures Plus	W-OTS+	3
WAMS#	Winternitz Abstracted Merkle Signature Sharp	W-OTS#	4

2.2.3 Field: Height

The height parameter `h` is a 1-byte value that determines how many OTS keys are coupled to a single public key. From the AMS perspective, the security of a public key degrades after being used more than `2^h` signature generations.

Specifically, `h` refers to the height of a merkle-tree whose leaves are a set of OTS public key hashes called the "batch". The merkle-root of the batch is called the "batch-root". A public key commits to a single batch. Signatures are signed using one of the OTS keys from the batch and never used again. The cardinality of a batch is `2^h`.

Whilst a public key can only be used for up to verify `2^h` distinct signature before security degradation, a private key can be used for `2^(64+h)` signature generations. This follows from the property that one private key can generate `2^64` unique public keys (and `2^h * 2^64 = 2^(64+h)`). Although a private key must be discarded beyond that number, it is for all practical purposes a reusable private key.

Choosing parameter `h` is left to the user as it's selection impacts the computational performance of the private key (but negligibly for the public key). Specifically, signing and verifying are negligibly impacted by `h` but generating keys and matching public keys to private keys has time complexity `O(h^2)`. If the user is able to replace their public key regularly, a low value of `0 <= h <= 8` is desirable. If a user plans to infrequently use their keys, a value `h=16` may be more appropriate in that expensive computations are done infrequently. The range `16 <= x <= 255` should be carefully considered, if at all.

NOTE Choosing `h=0` will result in an private key that signs a single message thus rendering AMS into a redundant OTS. It is permitted for elegancy of the scheme.

2.3 Public Key

The public key `K` is a many-time public key and defined as follows:

`K = v || h || C || B || Z || R`

Term	Description	Bits	Value Range
`C`	Key Code	64	UInt64
`B`	Batch Number	64	UInt64
`Z`	Spam Code	32	UInt32
`R`	Batch Root	U	hash digest

The public key is `(U/8 + 16)` bytes in length and can be used for `2^h` signature verifications before security begins to degrade. A single private key can derive up to `2^64` public keys by varying the `B` parameter.

2.3.1 Field: Key Code

The key code `C` is a 64-bit unsigned integer derived as:

`C = LastQWord( H^2( P ) )`

The key code is used to efficiently test if a public key derives from a private key `P`. Since the key code is a cryptographically random checksum with `2^64` possible values, it is unlikely to collide with other (genuine) keys and thus can be used to efficiently filter matching keys.

2.3.2 Field: Batch Number

A private key can generate up to `2^64` public keys each of which commits to `2^h` OTS public keys. The batch number identifies which batch a public key commits to. All batches are derived from the private key using the batch number as a generating nonce.

2.3.3 Spam Code

The Spam Code `Z` is a 32-bit unsigned integer checksum value derived as:

`Z = LastDWord(H(K'_0)))`

Term	Description
`K'_i`	The `i'th` OTS public key in the batch `B` (`i=0` in above)

The spam code works similarly to the key code except it is intended to thwart DoS attacks arising from deliberate key code collisions sent by a spammer. Without a spam code, a DoS attack could arise if a verifier is overwhelmed by a large list of public keys with key codes that (deliberately) collide with a known verifiers key code.

This can occur since calculating the batch-root is a computationally expensive process and a verifier can never determine if a public key is "cryptographically invalid". It can only determine if a public key derives from a private key (or not). This entails a batch-root calculation. The spam code is a mechanism to allow rapid filtering of such maliciously colliding invalid keys.

In the case where an attacker floods a verifier known good `B` and `Z` values, a verifier need only perform the expensive batch-root calculation once and cache the computed Public Key for `B`for future comparisons.

In the case where an attacker floods with varying `B` and `C` values, in an attempt to force the verifier to always evaluate the batch-root for `B`, precomputing/caching will not work since the range of values of `B` is too vast. In this scenario, the computation of the spam code is sufficient to discard this key, since the spammer cannot guess this code as it requires knowledge of the Private Key.

NOTE Knowledge of `C` or `Z` does not provide an attacker any computational advantage in a brute-force attack on `P`.

2.3.4 Field: Batch Root

The batch root is a merkle-root of the set of OTS public keys in the batch. It is defined as follows:

`R = MerkleRoot( H(K'_0), ..., H(K'_n) )`

Term	Description
`MerkleRoot`	The root of the hash-tree of the input leaf nodes
`K'_i`	The `i'th` OTS public key in the batch `B`
`h`	Tree Height parameter which determines batch size `2^m`
`n`	Index of last item in batch (which is always `2^h - 1`)

Here the batch-root commits to a set ephemeral OTS keys which will be used in signature generation and verification. Selection of the OTS key-pair is performed by the signer during the signing process and care should be taken to **never re-use** an OTS key.

REMARK In blockchain-based applications, this is achieved by using a strictly increasing Nonce field associated to the signer identity and stored in a public consensus database. The nonce is strictly incremented after every signature and part of transaction itself. Since the transaction update is ACID, it can be reliably used to for OTS key selection.

Merkle-tree roots and proofs follow standard merkle-tree constructions constructions widely documented and developed.

NOTE In the definition above, the leaf-set of of the merkle-tree comprise of public key hashes. If the OTS-layer of the AMS-algorithm passes up a public key hash, and matching CHF's are used, it need not be hashed again. Re-using the public key value is sufficient (as it is already a hash digest).

2.4 Signature Generation

For any message `M`, private key `P` and public key `K` a signature `S` is derived as follows:

```
algorithm AMS_Sign
  Input:
        M: a message to sign (arbitrarily long byte array)
        P: an AMS private key to used to sign
        B: the batch of OTS keys to use
        i: the index of the OTS key in the batch
  Output:
        S: an AMS signature
  Pseudo-Code:
     let v = P.Version
     let (P'_i, K'_i) = GenOTSKeys(P, B, i)
     let S' =  GenOTSSig( H(M), P'_i )
     S = v || h || i || K'_i || S' || GenMerkleProof(H(K'_i), i, R)
end algorithm
```

Term	Description	Bits
`v`	Version (from `P`)	`8`
`h`	Height (from `P`)	`8`
`i`	The index of the selected OTS key from the batch	`32`
`S'`	The OTS signature for the message	`OTS_SigBits`
`P'_i`	The OTS private key which derives `K'_i`	`OTS_PrivKeyBits`
`K'_i`	The `i'th` OTS public key from the batch	`OTS_PubKeyBits`
`R`	The batch-root of the batch which contains `K'_i`	`U`
`GenOTSKeys(x,y,z)`	OTS-layer function that generates the `z'th` OTS key-pair in batch `y` for private key `x`	
`GenOTSSig(x,y)`	OTS-layer function that generates an OTS signature of message-digest `x` using OTS private key `y`	
`GenMerkleProof(x,y,z)`	A standard merkle-tree function that generates a merkle-proof that `x` is `y'th` leaf of a merkle-tree with root `z`	
`OTS_SigBits`	Length of the OTS signature as returned by the OTS layer	
`OTS_PrivKeyBits`	Length of an OTS signature as returned by the OTS layer	
`OTS_PubKeyBits`	Length of an OTS signature as returned by the OTS layer. See Note below.	

Signatures are `(48 + OTS_PubKeyLenLength(S') + (h+1)*(U/8))` bytes in length, the bulk of which comprises of the OTS signature.

NOTE Whilst an AMS signature embeds an OTS public key, it's possible to compress the OTS public key in the OTS layer before passing it up to the AMS layer. This is done by using the hash of the public key (i.e. the "public key hash") rather than the public key itself. This works for many OTS algorithms since the signature verification process rebuilds the public key from the signature and thus it can verify against the public key hash instead. This optimization is employed in WAMS where the W-OTS public key is actually the W-OTS public key hash. Thus for any Winternitz parameter `w`, the "W-OTS public key" in the WAMS signature only consumes `U` bits (a significant optimization).

2.4.1 OTS Index

When signing. the selection of `i` is performed by the signer and care should be taken to not re-use a previously used one-time key. In blockchain-based applications, this is achieved by using the strictly increasing Nonce field from signing identity object stored in a public consensus database. Since this does not require local database updates, and the Nonce is always updated across a consensus database after a signed transaction is confirmed, there is no risk of accidental key reuse arising from local state corruption. Signature re-use could still exist but this would arise from a bug or attack to the client code (no different to traditional cryptographic schemes). When used in this manner, the AMS scheme can be considered "practically stateless", however it is not strictly so.

In non-blockchain/DLT applications, care should be taken to remember the index of the last OTS signature so as not to re-use. Note, if an attacker is able to trick the code into re-using an OTS key, it's security could be totally compromised after a few re-uses.

2.5 Signature Verification

```
algorithm AMS_Verify:
  Input:
        M: the message being verified (arbitrary byte array)
        S: the AMS signature of M
        K: the AMS public key used to verify S
  Output: Boolean
  Pseudo-Code:
    let size0 = U/8                              ; byte size of a CHF digest
    let size1 = OTS_SigLen/8                     ; byte size of an OTS signature
    let size2 = OTS_PubKeyLen/8                  ; byte size of an OTS public key (hash)

    let reader = byte stream reader for S        ; a little-endian byte reader
    let v = reader.ReadByte                      ; AMS version
    let h = reader.ReadWord                      ; height
    let i = reader.ReadUInt                      ; OTS key index
    let PKH = reader.ReadBytes(size2)            ; OTS public key (hash)
    let S' = reader.ReadBytes(size1)             ; OTS signature
    let MP = reader.ReadBytes(h * size0)         ; merkle-proof
    let R = K.R                                  ; batch root
    Result = (v == K.Version) AND VerMerkleProof(MP, PKH, i, 2^h, R) AND VerOTSSig(S', H(M), PKH)
end algorithm
```

Term	Description
`VerOTSSig(x, y, z)`	An OTS-layer function that returns true iff `x` is an OTS signature of message-digest `y` which verifies with OTS public key (hash) `z`, otherwise returns false.

Term	Description
`VerMerkleProof(u, v, w, x, y)`	An OTS-layer function function that returns true iff `u` is the merkle-proof that `v` is the `w'th` leaf in a merkle-tree whose leaves are a set of cardinality `x` and whose merkle-root is `y`.

The `VerMerkleProof` term ensures that the OTS key used by the signature was committed to by the public key and the `VerOTSSig` term verifies the OTS signature verifies to that OTS key. With these two steps, the verifier has determined the AMS signature contains a valid OTS signature and that the OTS public key in the signature was committed to by the AMS public key. Since only the bearer of the AMS private key can know the OTS private key, it follows the AMS signature was signed by the bearer of the AMS private key.

NOTE it is desirable that the merkle-tree construction employed within AMS should not require existence proofs to contain direction flags when traversing the tree. Instead these directions ought to be implicit and inferred from the index of the leaf-node being traversed from and the cardinality of the set of leaf nodes. Merkle "existence proof" should only ever contain the minimal set of parent-node hashes required to evaluate the proof.

3. Reference Implementation

This section contains snippets for the full reference implementation[3]. The reference implementation is part of the PQC library within the Hydrogen Framework[4].

3.1 OTS Interface

```
public interface IOTSAlgorithm {
    OTSConfig Config { get; }
    void SerializeParameters(Span<byte> buffer);
    void ComputeKeyHash(byte[,] key, Span<byte> result);
    byte[,] SignDigest(byte[,] privateKey, ReadOnlySpan<byte> digest);
    bool VerifyDigest(byte[,] signature, byte[,] publicKey, ReadOnlySpan<byte>
digest);
    OTSKeyPair GenerateKeys(ReadOnlySpan<byte> seed);
}

public static class IOTSAlgorithmExtensions {
    public static byte[] ComputeKeyHash(this IOTSAlgorithm algo, byte[,] key) {
        var result = new byte[algo.Config.DigestSize];
        algo.ComputeKeyHash(key, result);
        return result;
    }
}
```

3.2 AMS Implementation

```
public class AMS : DigitalSignatureSchemeBase<AMS.PrivateKey, AMS.PublicKey> {
    public const int MaxHeight = 20;
    public const byte Version = 1;
    private readonly IOTSAlgorithm _ots;

    public AMS(AMSOTS ots)
        : this(InstantiateOTSAlgorithm(ots)) {
    }

    public AMS(AMSOTS ots, int h)
        : this(InstantiateOTSAlgorithm(ots), h) {
    }

    public AMS(IOTSAlgorithm algorithm)
        : this(algorithm, Configuration.DefaultHeight) {
    }

    public AMS(IOTSAlgorithm algorithm, int h)
        : this (algorithm, new Configuration(algorithm.Config, h)) {
    }

    public AMS(IOTSAlgorithm algorithm, Configuration config)
        : base(algorithm.Config.HashFunction) {
        Config = config;
        _ots = algorithm;
        Traits = Traits & DigitalSignatureSchemeTraits.PQC;
    }

    public Configuration Config { get; }

    public override IIESAlgorithm IES => throw new NotSupportedException("PQC
algorithms have no known IES algorithms");

    public override bool TryParsePublicKey(ReadOnlySpan<byte> bytes, out PublicKey
publicKey)
        => PublicKey.TryParse(bytes, _ots.Config.HashFunction, out publicKey);

    public override bool TryParsePrivateKey(ReadOnlySpan<byte> bytes, out PrivateKey
privateKey)
        => PrivateKey.TryParse(bytes, _ots.Config.HashFunction, out privateKey);

    public override PrivateKey GeneratePrivateKey(ReadOnlySpan<byte> secret) {
        if (secret.Length != 32)
            throw new ArgumentException("Must be 256-bit value", nameof(secret));
        var rawBytes = new byte[64];
        Array.Fill(rawBytes, (byte)0);
        rawBytes[0] = Version;
        EndianBitConverter.Little.WriteTo((byte)_ots.Config.AMSID, rawBytes, 1);
        rawBytes[3] = (byte)Config.H;
        _ots.SerializeParameters(rawBytes.AsSpan(4, 28));
        secret.CopyTo(rawBytes.AsSpan(^32));
        return new PrivateKey(rawBytes, _ots.Config.HashFunction);
    }

    public override PublicKey DerivePublicKey(PrivateKey privateKey, ulong
signerNonce) {
        var batchLength = 1U << privateKey.Height;
        var batchNo = signerNonce / batchLength;
        return DerivePublicKeyForBatch(privateKey, batchNo, true);
    }

    public PublicKey DerivePublicKeyForBatch(PrivateKey privateKey, ulong batchNo,
bool rememberBatch = false) {
        var batch = CalculateBatch(privateKey, batchNo, out var spamCode);
        var rawPubKey = Tools.Array.Concat<byte>(
            EndianBitConverter.Little.GetBytes((uint)privateKey.KeyCode),
            EndianBitConverter.Little.GetBytes((ulong)batchNo),
            EndianBitConverter.Little.GetBytes((uint)spamCode),
            batch.Root
        );
        if (rememberBatch) {
            var publicKeyWithBatch = new PublicKeyWithBatch(rawPubKey, batch);
            privateKey.RememberDerivedKey(publicKeyWithBatch);
        }
        return new PublicKey(rawPubKey);
    }

    public override bool IsPublicKey(PrivateKey privateKey, ReadOnlySpan<byte>
publicKeyBytes) {
        var batchNo = PublicKey.ExtractBatchNo(publicKeyBytes);
        return
            privateKey.KeyCode == PublicKey.ExtractKeyCode(publicKeyBytes) &&
            CalculateSpamCode(privateKey, batchNo) ==
PublicKey.ExtractSpamCode(publicKeyBytes) &&
            DerivePublicKeyForBatch(privateKey,
batchNo).RawBytes.AsSpan().SequenceEqual(publicKeyBytes);
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public byte[] Sign(PrivateKey privateKey, ReadOnlySpan<byte> message, ulong
batchNo, int otsIndex) {
        var messageDigest = CalculateMessageDigest(message);
        return SignDigest(privateKey, messageDigest, batchNo, otsIndex);
    }

    public override byte[] SignDigest(PrivateKey privateKey, ReadOnlySpan<byte>
messageDigest, ulong signerNonce) {
        var (batchNo, otsIndex) = GetOTSIndex(privateKey.Height, signerNonce);
        return SignDigest(privateKey, messageDigest, batchNo, otsIndex);
    }
    public byte[] SignDigest(PrivateKey privateKey, ReadOnlySpan<byte>
messageDigest, ulong batchNo, int otsIndex) {
        var builder = new ByteArrayBuilder();
        // Append header
        builder.Append(privateKey.Height);
        builder.Append(EndianBitConverter.Little.GetBytes((ushort)otsIndex));

        // Get/Calc the OTS batch
        if (!privateKey.DerivedKeys.TryGetValue(batchNo, out var
publicKeyWithBatch)) {
            DerivePublicKeyForBatch(privateKey, batchNo, true);
            publicKeyWithBatch = privateKey.DerivedKeys[batchNo];
        }
        var otsPubKey =
            Config.OTS.UsePublicKeyHashOptimization ?
            publicKeyWithBatch.Batch.GetValue(MerkleCoordinate.LeafAt(otsIndex)):
            this.GetOTSKeys(privateKey, batchNo, otsIndex).PublicKey.AsFlatSpan();

        Debug.Assert(otsPubKey.Length == Config.OTS.PublicKeySize.Length *
Config.OTS.PublicKeySize.Width);
        builder.Append(otsPubKey);

        // Derive the individual private key again
        // NOTE: possibility to optimize here if we want to cache ephemeral OTS
private key, but large in memory
        var otsKey = GetOTSKeys(privateKey, batchNo, otsIndex);

        // Perform the OTS sig
        var otsSig = _ots.SignDigest(otsKey.PrivateKey,
messageDigest).ToFlatArray();
        Debug.Assert(otsSig.Length == _ots.Config.SignatureSize.Length *
_ots.Config.SignatureSize.Width);
        builder.Append(otsSig);

        // Append merkle-existence proof of pubKey in Batch (will always be 2^h
hashes)
        var authPath =
publicKeyWithBatch.Batch.GenerateExistenceProof(otsIndex).ToArray();
        foreach (var bytes in authPath) {
            builder.Append(bytes);
        }

        var sig = builder.ToArray();
        return sig;
    }

    public override bool VerifyDigest(ReadOnlySpan<byte> signature,
ReadOnlySpan<byte> digest, ReadOnlySpan<byte> publicKey) {
        Guard.Argument(IsWellFormedSignature(signature), nameof(signature), "Not a
valid AMS signature");
        Guard.Argument(digest.Length == _ots.Config.DigestSize, nameof(digest),
$"Message digest must be { _ots.Config.DigestSize } bytes");
```

```
        var reader = new ByteSpanReader(EndianBitConverter.Little);
        var height = reader.ReadByte(signature);
        var otsIndex = reader.ReadUInt16(signature);
        var otsPubKey = reader.ReadBytes2D(signature,
_ots.Config.PublicKeySize.Length , _ots.Config.PublicKeySize.Width);
        var otsSig = reader.ReadBytes2D(signature, _ots.Config.SignatureSize.Length,
_ots.Config.SignatureSize.Width);
        var proof = new byte[height][];
        for (var i = 0; i < proof.Length; i++)
            proof[i] = reader.ReadBytes(signature, _ots.Config.DigestSize);

        // OTS Key must exist in batch
        var otsPubKeyHash = Config.OTS.UsePublicKeyHashOptimization ?
otsPubKey.AsFlatSpan() : _ots.ComputeKeyHash(otsPubKey);
        if (!MerkleMath.VerifyExistenceProof(_ots.Config.HashFunction,
PublicKey.ExtractBatchRoot(publicKey).ToArray(), MerkleSize.FromLeafCount(1 <<
height), MerkleCoordinate.LeafAt(otsIndex), otsPubKeyHash, proof))
            return false;

        // OTS sig must be valid
        return _ots.VerifyDigest(otsSig, otsPubKey, digest);
    }

    public bool IsWellFormedSignature(ReadOnlySpan<byte> signature) {
        if (signature == null || signature.Length == 0)
            return false;
        var h = signature[0];
        return signature.Length == (
            3
            + (_ots.Config.PublicKeySize.Length * _ots.Config.PublicKeySize.Width)
            + (_ots.Config.SignatureSize.Length * _ots.Config.SignatureSize.Width)
            + h * _ots.Config.DigestSize);
    }

    private IMerkleTree CalculateBatch(PrivateKey privateKey, ulong batchNo, out
uint spamCode) {
        var batchSize = 1 << privateKey.Height;
        var batchLeafs = new byte[batchSize][];
        Parallel.For(0, batchSize, i => {
            batchLeafs[i] = GetOTSKeys(privateKey, batchNo, i).PublicKeyHash.Value;
        });
        spamCode = CalculateSpamCode(batchLeafs[0]);
        var merkleTree = new SimpleMerkleTree(_ots.Config.HashFunction);
        merkleTree.Leafs.AddRange(batchLeafs);
        return merkleTree;
    }

    private uint CalculateSpamCode(PrivateKey privateKey, ulong batchNo)
        => CalculateSpamCode(GetOTSKeys(privateKey, batchNo,
0).PublicKeyHash.Value);

    private uint CalculateSpamCode(ReadOnlySpan<byte> wotsKey0)
```

```
        => EndianBitConverter.Little.ToUInt32(wotsKey0.Slice(^4));

    private OTSKeyPair GetOTSKeys(PrivateKey privateKey, ulong batchNo, int index)
=>
        _ots.GenerateKeys(Tools.Array.Concat<byte>
(EndianBitConverter.Little.GetBytes((uint)index),
EndianBitConverter.Little.GetBytes((ulong)batchNo), privateKey.RawBytes));

    private (ulong batchNo, int otsIndex) GetOTSIndex(int height, ulong signerNonce)
{
        var batchLength = 1U << height;
        return (signerNonce / batchLength, (int)(signerNonce % batchLength));
    }

    private static IOTSAlgorithm InstantiateOTSAlgorithm(AMSOTS ots) {
        switch (ots) {
            case AMSOTS.WOTS:
                return new WOTS(WOTS.Configuration.Default.W, true);
            case AMSOTS.WOTS_Sharp:
                return new WOTSSharp(WOTSSharp.Configuration.Default.W, true);
            default:
                throw new NotSupportedException(ots.ToString());
        }
    }

    public abstract class Key : IKey {

        protected Key(byte[] immutableRawBytes) {
            RawBytes = immutableRawBytes;
        }

        public readonly byte[] RawBytes;

        public override bool Equals(object obj) {
            if (obj is Key key) {
                return Equals(key);
            }
            return false;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected bool Equals(Key other) {
            return Equals(RawBytes, other.RawBytes);
        }

        public override int GetHashCode() {
            return (RawBytes != null ? RawBytes.GetHashCode() : 0);
        }

        #region IKey
        byte[] IKey.RawBytes => RawBytes;
        #endregion
    }

    public class PrivateKey : Key, IPrivateKey {

        public readonly byte Version;
        public readonly byte Height;
        public readonly uint KeyCode;
        public readonly Dictionary<ulong, PublicKeyWithBatch> _derivedKeys;

        internal PrivateKey(byte[] immutableRawBytes, CHF chf)
            : base(immutableRawBytes) {
            Version = immutableRawBytes[0];
            Guard.Argument(Version == AMS.Version, nameof(immutableRawBytes),
"Unrecognized version");

            Height = immutableRawBytes[1];
            Guard.Argument(0 <= Height && Height <= MaxHeight,
nameof(immutableRawBytes), "Unsupported key height");
            KeyCode = CalculateKeyCode(immutableRawBytes, chf);
            _derivedKeys = new Dictionary<ulong, PublicKeyWithBatch>();
        }

        internal void RememberDerivedKey(PublicKeyWithBatch publicKey) =>
_derivedKeys[publicKey.BatchNo] = publicKey;

        public IReadOnlyDictionary<ulong, PublicKeyWithBatch> DerivedKeys =>
_derivedKeys;

        public static bool TryParse(ReadOnlySpan<byte> rawBytes, CHF chf, out
PrivateKey privateKey) {
            var version = rawBytes[0];
            if (version != AMS.Version) {
                privateKey = null;
                return false;
            }
            var height = rawBytes[1];
            if (height > MaxHeight) {
                privateKey = null;
                return false;
            }
            privateKey = new PrivateKey(rawBytes.ToArray(), chf);
            return true;
        }

        private static uint CalculateKeyCode(ReadOnlySpan<byte> privateKeyRawBytes,
CHF chf)
            => EndianBitConverter.Little.ToUInt32(Hashers.Iterate(chf,
privateKeyRawBytes, 2).AsSpan(^4));

    }

    public class PublicKey : Key, IPublicKey {
        public readonly ulong BatchNo;
        public readonly uint KeyCode;
        public readonly uint SpamCode;
        public readonly byte[] BatchRoot;

        internal PublicKey(byte[] immutableRawBytes)
            : base(immutableRawBytes) {
            KeyCode = ExtractKeyCode(immutableRawBytes);
            BatchNo = ExtractBatchNo(immutableRawBytes);
            SpamCode = ExtractSpamCode(immutableRawBytes);
            BatchRoot = ExtractBatchRoot(immutableRawBytes).ToArray();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static uint ExtractKeyCode(ReadOnlySpan<byte> publicKeyRawBytes)
            => EndianBitConverter.Little.ToUInt32(publicKeyRawBytes, 0);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong ExtractBatchNo(ReadOnlySpan<byte> publicKeyRawBytes)
            => EndianBitConverter.Little.ToUInt64(publicKeyRawBytes, 4);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static uint ExtractSpamCode(ReadOnlySpan<byte> publicKeyRawBytes)
            => EndianBitConverter.Little.ToUInt32(publicKeyRawBytes, 12);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ReadOnlySpan<byte> ExtractBatchRoot(ReadOnlySpan<byte>
publicKeyRawBytes)
            => publicKeyRawBytes.Slice(16);

        public static bool TryParse(ReadOnlySpan<byte> rawBytes, CHF chf, out
PublicKey publicKey) {
            if (rawBytes.Length != Hashers.GetDigestSizeBytes(chf) + 16) {
                publicKey = null;
                return false;
            }
            publicKey = new PublicKey(rawBytes.ToArray());
            return true;
        }
    }

    public class PublicKeyWithBatch : PublicKey {
        public readonly IMerkleTree Batch;

        internal PublicKeyWithBatch(byte[] immutableRawBytes, IMerkleTree batch)
            : base(immutableRawBytes) {
            Batch = batch;
        }

    }

    public sealed class Configuration : ICloneable {
        public const int DefaultHeight = 8;
        public readonly int H;
        public readonly OTSConfig OTS;

        public Configuration(OTSConfig otsConfig) : this(otsConfig, 8) {
        }

        public Configuration(OTSConfig otsConfig, int h) {
            Guard.ArgumentInRange(h, 0, AMS.MaxHeight, nameof(h));
            OTS = (OTSConfig)otsConfig.Clone();
            H = h;
        }

        public Configuration Clone() => new Configuration(OTS, H);

        object ICloneable.Clone() => Clone();

    }

}
```

4. References

1. Ralph Merkle. "Secrecy, authentication and public key systems / A certified digital signature". Ph.D. dissertation, Dept. of Electrical Engineering, Stanford University, 1979. Url: http://www.merkle.com/papers/Certified1979.pdf ↩

2. IRTF. "XMSS: eXtended Merkle Signature Scheme". Accessed: 2020-07-01, URL: https://tools.ietf.org/html/rfc8391 ↩

3. Sphere 10 Software. PQC Library. Accessed 2023-05-09, Url: https://github.com/Sphere10/Hydrogen/tree/master/src/Hydrogen/Crypto/PQC ↩

4. Sphere 10 Software. Hydrogen Framework. Accessed 2023-05-09, Url: https://github.com/Sphere10/Hydrogen ↩