DV Labs

Towards Post-Quantum Distributed Validators on Ethereum

Abstract: With recent advances in quantum computing, post-quantum cryptographic algorithms are being actively deployed in real-world applications. For Ethereum, a transition to post-quantum cryptography would require replacing many primitives, including the BLS12-381 signature schemes used by validators on the beacon chain. Because BLS leverages its bilinear pairing property to aggregate multiple validator signatures, enabling both performance improvements and space savings, its replacement presents a particular challenge. Furthermore, the bilinearity of the pairing function also enables straightforward threshold signatures, which are fundamental to distributed validator solutions. The objective of this report is to review the current state of post-quantum signature algorithms and identify those most suitable for the Ethereum beacon chain, particularly focusing on their support for signature aggregation and threshold functionality.

Contents

January 29, 2025

1 Introduction

If a cryptographically relevant quantum computer is built, Shor's algorithm will pose serious threats to traditional public key cryptosystems based on large number factorization and (elliptic curve) discrete logarithm problems, such as RSA and ECDSA. As a response, cryptographers are developing new algorithms that offer security even against an attacker equipped with quantum computers, denoted as post-quantum cryptography (PQC). There are several standardization processes ongoing, notably by the one NIST which has already published a set of standards: ML-KEM [?] as key encapsulation mechanism along with ML-DSA [?], SLH-DSA [?] and FN-DSA, to be published soon, as signature schemes. Because ML-DSA and FN-DSA are both lattice-based, NIST has sollicited the submission of additional signatures schemes to expand its PQC signature portfolio[1]. Unfortunately, the current post-quantum signature schemes selected by NIST for standardization do not inherently support advanced functionalities such as signature aggregation and/or threshold signing. Signature aggregation is commonly used in blockchain systems as this powerful feature allows to compress many signatures into a short aggregate, shrinking the storage space and speeding-up the verification time. Ethereum leverages takes advantage of aggregate signatures in its consensus layer thanks to the BLS signature scheme [?]. On top of intrinsically supporting signature/public key aggregation, BLS is straightforward to be turned into a threshold signature scheme when combined with Shamir Secret Sharing. An important observation in the case of BLS is that aggregated and/or threshold BLS signatures are indistinguishable from raw ones, all being points on the same elliptic curve. This allows to build efficient distributed validator solutions, such as the charon middleware[2] which operates in a totally transparent manner from a consensus client point of view. However, because BLS is based on elliptic curve pairing, it would not provide enough security against quantum adversaries. To address this concern, the Ethereum foundation recently introduced a family of hash-based signature schemes as post-quantum atlernatives to BLS [?]. The main idea behind their design is to aggregate hash-based signatures using post-quantum succinct non-interactively argument of knowledge (pqSNARK) systems.While this seems to be a promising alternative, it might have considerable impacts on distributed validators solutions which currently take advantage of the homomorphic properties from BLS. The goal of this document is to identify the challenges that could arise from such a transition and discuss the potential solutions to address them.

2 Aggregate Hash-Based Signatures using SNARKs

2.1 Hash-based signatures

As their name suggests, hash-based signature schemes rely on hash functions as their core primitive. In contrast to public key cryptosystems, there is no strong evidence that symmetric cryptography, including hash functions, would be significantly impacted by quantum computers. Although recommendations on symmetric cryptography may vary between cybersecurity agencies[3], hash-based signatures are seen as a conservative choice for post-quantum security given their well-understood security. The classical approach to build hash-based signatures is to combine many one-time signature (OTS) key pairs into a Merkle tree [?] whose root serves as the many-time public key. To provide a concrete example, we hereafter introduce the Winternitz OTS (WOTS) scheme.

[1] https://csrc.nist.gov/projects/pqc-dig-sig

[2] https://github.com/ObolNetwork/charon

[3] ANSSI recommends at least the same security as AES-256 and SHA2-384 for block ciphers and hash functions, respectively, whereas NIST, NCSC and BSI recommend AES-128 and SHA-256.

Winternitz OTS. WOTS is parameterized by two values:

- the Winternitz parameter w, being a power of 2.
- a n-bit hash function H such that $n = vw$

To generate an OTS key pair, one randomly generates v n-bit secret keys $sk_0, \cdots, sk_{v-1}$ and derives the corresponding public keys using an hash chain of length $2^w - 1$ (*i.e.*, $pk_i = H^{2^w-1}(sk_i)$). To sign a message m, a checksum over m is appended to it before hashing. The n-bit output is then divided into v w-bit chunks $c_0, \cdots, c_{v-1}$ and the signature consists of $\sigma = \sigma_0, \cdots, \sigma_{v-1}$ where $\sigma_i = H^{c_i}(sk_i)$. To verify a signature, one checks that $H^{w-1-c_i}(\sigma_i) = pk_i$ for $i \in \{0, \cdots, v-1\}$.

Merkle tree. To build a many-time signature scheme from WOTS, one can combine multiple key pairs with a binary tree where each node is the hash of its children, commonly referred to as Merkle tree. For a height parameter h, such a tree is built from 2^h leaves $l_0, \cdots, l_{2^h-1}$, each being the hash of a WOTS public key (*i.e.*, $l_i = H(pk_{i_0}, \cdots, pk_{i_{v-1}})$). The root constitutes the many-time public key and commits to all OTS public keys. Note that to reduce memory requirements in practice, it is recommended to generate the WOTS secret keys using a pseudorandom function (PRF) rather than using a random number generator [**?**]. To sign the ith message, the signer uses the ith OTS secret key and includes the Merkle path of the corresponding public key in the signature. To verify the signature, the verifier computes the public key from WOTS signature and then, thanks to the Merkle path, verifies that its digest is indeed the leaf at position i. This introduces the concept statefulness: because security depends on the unique usage of each OTS key pair, it is crucial to keep track of which keys have already been used.

2.2 SNARK-based aggregation

The idea behind SNARK-based aggregation is for an aggregator to turn individual signatures, possibly over different messages, into a SNARK proof attesting their validity. Note that this principle can be used to thresholdize a signature scheme: given a k-of-n setting, the aggregator can generate a proof attesting that it verified k distinct signatures over the same message and that signers are part of the quorum. A valuable feature of this approach is its non-interactiveness: the aggregator only needs to collect individual signatures in order to compute the proof, without any additional communication. Combining such a construction with hash-based signatures has been first explored by Khaburzaniya *et al.* [**?**], using WOTS with 1-bit chunks (instantiated with the Rescue-Prime hash function) along with STARKs. To complement this research, the work from Drake *et al.* [**?**] does not focus on a specific hash-based signature scheme but explores a variety of tradeoffs by introducing a generalized variant of XMSS [**?**] and providing security proofs that hold for all its instantiations. Notably, their security proofs do not model hash functions as random oracles and rely on standard model properties instead, such as preimage/collision resistance, providing concrete security targets.

3 Thresholdizing XMSS

The downside of building a threshold hash-based signature by leveraging a SNARK system as mentioned above is that the aggregation of threshold signatures would not be straightforward (since threshold signatures are proofs instead of raw hash-based signatures). In the case of the Beacon chain where threshold signatures occur *before* aggregation duties, it is imperative for distributed validator middlewares to output signatures that can be aggregated according to the protocol. Therefore, this section focuses on constructions which lead to threshold Winternitz signatures that are indistinguishable from non-threshold ones, as in BLS.

3.1 Distributed hash-based signatures with Boolean shares.

Distributed variants of hash-based signatures, including XMSS, have been explored by Kelsey, Lang and Lucks in [?] where they introduce n-of-n and k-of-n threshold signature schemes which rely on Boolean shares. For the n-of-n setting, a trusted dealer starts from an existing Merkle tree and splits each WOTS secret key $\mathsf{sk_i}$ by generating n random values $\mathsf{r}_i^0, \cdots, \mathsf{r}_i^{n-1}$ to compute $\mathsf{r}_i^h = \mathsf{r}_i^0 \oplus \mathsf{r}_i^1 \oplus \cdots \oplus \mathsf{r}_i^{n-1} \oplus \mathsf{sk}_i$. This introduces an additional party called the *helper* whose role is to store and provide the relevant helper shares whenever required. That way, to produce a WOTS using for sk_i, each party can sign independently using its Boolean key share assuming the aggregator has access to the helper share r_i^h. Note that the trusted dealer not only has to produce helper shares for WOTS secret keys, but also for each Merkle paths, inducing high memory requirement on the helper. To minimize memory usage for the parties, the key shares are actually generated pseudorandomly using a PRF along with a secret seed as detailed in Algorithm 1. To turn their n-of-n scheme into a k-of-n threshold scheme, they propose to instantiate a Merkle tree that contains keys for all possible $\binom{n}{k}$ quorums. Beyond complexity, this increases the height of the Merkle tree and hence the signature size as well as the memory requirements for the helper. Because their designs are incompatible with distributed key generation (DKG) methods, we investigate other solutions instead.

Algorithm 1: Split a Merkle tree of WOTS keys into distributed key shares for n-of-n signatures, according to [?].

Input parameters:

- Merkle tree built out of a n-bit hash function H and 2^h WOTS secret keys $\mathsf{sk}_0, \cdots, \mathsf{sk}_{2^h-1}$ to sign $n = vw$-bit messages (*i.e.*, $\mathsf{sk}_i = (\mathsf{sk}_{i,0}, \cdots, \mathsf{sk}_{i,v-1})$).
- A pseudorandom function $\mathsf{PRF}_K(x, l)$ parametrized by a k-bit key K which takes as input a seed x along with the output bit length l.
- A set of distributed parties $\mathcal{P}$.

Output parameters:

- Secret keys key_p for each party $p \in \mathcal{P}$.
- Helper shares $\mathsf{sk}^h_{i,j}$ and path^h_i for $i \in \{0, \cdots, 2^h - 1\}$ and $j \in \{0, \cdots, v - 1\}$.

```
foreach $p \in \mathcal{P}$ do                    // picks secrets at random for each party
    $\mathsf{key}_p \xleftarrow{\$} \{0,1\}^k$
end foreach
for $i = 0$ to $2^h - 1$ do                          // builds Merkle path helper shares
    $\mathsf{path}^h_i \leftarrow \mathsf{path}_i$
    foreach $p \in \mathcal{P}$ do
        $\mathsf{path}^h_i \leftarrow \mathsf{path}^h_i \oplus \mathsf{PRF}_{\mathsf{key}_p}((\mathsf{domain}_{\mathsf{path}}, i), nh)$
    end foreach
end for
for $i = 0$ to $2^h - 1$ do                            // builds WOTS key helper shares
    for $j = 0$ to $v - 1$ do                          // for each secret key component
        for $c = 0$ to $2^w - 1$ do                    // for each w-bit chunk to sign
            $\mathsf{sk}^h_{i,j}[c] \leftarrow H^c(\mathsf{sk}_{i,j})$
            foreach $p \in \mathcal{P}$ do
                $\mathsf{sk}^h_{i,j}[c] \leftarrow \mathsf{sk}^h_{i,j}[c] \oplus \mathsf{PRF}_{\mathsf{key}_p}((\mathsf{domain}_{\mathsf{key}}, i, j, c), n)$
            end foreach
        end for
    end for
end for
```

3.2 Leveraging Shamir Secret Sharing

Turning WOTS into a threshold scheme presents no significant challenges as one can simply apply a secret sharing scheme on the secret keys so that messages can be partially signed non-interactively. However, the difficulty arises when Merkle trees come into play: computing Merkle paths from secret key shares requires to process all hash function calls in a multi-party computation (MPC) fashion. This would be highly impractical in low latency scenarios such as performing validator duties on Ethereum, as the time to produce a threshold signature could not fit within an epoch (see *e.g.* the work from Cozzo and Smart [?] which estimates around 85 minutes to compute a threshold SPHINCS+ signature with SHA-3 as the underlying hash function). A possible workaround is to store Merkle paths along with WOTS secret keys so that it is not necessary to recompute them on-the-fly.

Bear in mind that avoiding any MPC hash invocation implies that WOTS secret keys cannot be generated pseudo-randomly from a secret seed shared amongst participants, further increasing memory consumption.

Still, even if all parties store their own WOTS secret key shares and corresponding Merkle paths, they cannot sign non-interactively. Indeed, XMSS not being a deterministic signature scheme, the signers have to agree on a random number to be included in the randomized message digest calculation.