Towards Threshold Hash-Based Signatures for Post-Quantum Distributed Validators

Alexandre Adomnicăi

DV Labs, alexandre@dvlabs.tech

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 Ethereum foundation recently introduced hash-based signature schemes as post-quantum alternatives to BLS. In this research, we study the practical challenges of deploying such schemes in a distributed manner.

Keywords: MPC · Hash-based signatures

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 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 which lends itself to Distributed Validator Technology (DVT) [?]. 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 DVT middleware solutions, such as the `charon`[2] middleware, which operates in a totally transparent manner from a consensus client point of view. However, because BLS is based on elliptic curve

[1] https://csrc.nist.gov/projects/pqc-dig-sig
[2] https://github.com/ObolNetwork/charon

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 would have considerable impacts on distributed validators solutions which currently rely on the homomorphic properties of BLS to leverage threshold signatures. 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.

Table 1: Properties comparison between BLS and hash-based signature (HBS) schemes.

	BLS	HBS
post-quantum secure	✗	✓
native aggregation support	✓	✗
non-interactive threshold signing	✓	✗
deterministic	✓	✗

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.

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 hash chains 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 $H^{2^w-1-c_i}(\sigma_i) = pk_i$ 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

[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.

$l_0, \cdots, l_{2^h-1}$, 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 Towards threshold 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 it has to be done for each component of the secret key: assuming a WOTS scheme to sign $n = vw$-bit messages, it means that each WOTS key requires $v2^w - 1$ helper shares. Furthermore, the trusted dealer also needs to provide the helper with shares for each Merkle paths, leading to high memory requirement for the helper overall. To minimize memory usage for the parties, the key shares are actually generated pseudorandomly using a PRF 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. Overall, this approach comes with several limitations from a DVT perspective. First, it is incompatible with distributed key generation (DKG) algorithms since a trusted dealer is required to split the key into multiple shares. While supporting DKG is not a necessary prerequisite for distributed validators, it is a valuable feature as it ensures that the private key is never known in its entirety by any single party. Second, and more importantly, the helper role contradicts with the nature of DVT by introducing a single point of failure which affects decentralization. Therefore, we investigate alternative solutions that could overcome these weaknesses.

3.2 Leveraging secret sharing

Another approach is to leverage a secret sharing scheme to split WOTS secret keys into shares distributed among participants. However, it requires to jointly compute all hash function calls in a multi-party computation (MPC) setting. This can be very challenging in practice, especially in low latency scenarios such as performing validator duties on Ethereum, as the time to produce a threshold signature may largely exceed the requirements (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 could be to store all secret keys calculated during key generation, so that it is possible to sign messages efficiently without any online MPC calculation. However, as highlighted in Table2, this could lead to unrealistic memory usage as it requires to precompute secret keys for every possible chunk value, and this for all tree leafs. Since tree nodes are not considered secret material thanks to the preimage resistance of the underlying hash function, a more pragmatic approach would be to only store the plain (*i.e.*, non-shared) leafs value so that signers will be able to calculate Merkle paths in a non-distributed manner when generating signatures. Nevertheless, in the case of DKGs, this still requires computing all hash function calls over MPC at key generation time. A comprehensive performance analysis of MPC hash functions is necessary to evaluate the timing constraints of DKG setups and to identify practical time-memory tradeoffs for signature generation.

4 Hash functions over MPC

Traditional hash functions such as SHA3 operate over binary fields to enable efficient implementations in both hardware and software on a wide range of platforms. However, they lead to poor performance when employed within advanced cryptographic protocols such as MPC. This is mainly due to the fact that traditional schemes are designed to minimize their overall gate count without minimizing specifically nonlinear gates[4] which require communication between parties in an MPC setting, unlike linear gates that can be computed locally. The overload induced by these communications is such that it can constitute the bottleneck in MPC protocols, as highlighted by an attempt to thresholdize PQC signatures schemes [?]. In response, new primitives with design constraints finely

[4]They are actually symmetric designs that aim at minimizing the number of nonlinear gates for efficient software masked implementations against side-channel attacks, see for instance [?].

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 $j \in \{0, \cdots, v-1\}$.

`// picks secrets at random for each party`
foreach $p \in \mathcal{P}$ **do**
  $\mathsf{key}_p \xleftarrow{\$} \{0,1\}^k$
end foreach

`// builds Merkle path helper shares`
for $i = 0$ **to** $2^h - 1$ **do**
  $\mathsf{path}^h_i \leftarrow \mathsf{path}_i$
  foreach $p \in \mathcal{P}$ **do**
    $\mathsf{path}^h_i \leftarrow \mathsf{path}^h_i \oplus \mathsf{PRF}_{\mathsf{key}_p}\big((\mathsf{domain}_{\mathsf{path}}, i), nh\big)$
  end foreach
end for

`// builds WOTS key helper shares`
for $i = 0$ **to** $2^h - 1$ **do** `// for each WOTS secret key`
  for $j = 0$ **to** $v - 1$ **do** `// for each key component`
    for $c = 0$ **to** $2^w - 1$ **do** `// for each w-bit chunk`
      $\mathsf{sk}^h_{i,j}[c] \leftarrow H^c(\mathsf{sk}_{i,j})$
      foreach $p \in \mathcal{P}$ **do**
        $\mathsf{sk}^h_{i,j}[c] \leftarrow \mathsf{sk}^h_{i,j}[c] \oplus \mathsf{PRF}_{\mathsf{key}_p}\big((\mathsf{domain}_{\mathsf{key}}, i, j, c), n\big)$
      end foreach
    end for
  end for
end for

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

tuned for advanced cryptographic protocols have emerged, known as *arithmetization-oriented* primitives. They usually operate over $\mathbb{F}_p$ with p prime, making them natively compatible with linear secret sharing schemes, and rely on multiplications for nonlinear operations. Among them, Poseidon [?] has found its place into many Ethereum applications thanks to its efficiency in verifiable computing and its successor Poseidon2 [?] is currently being considered for Ethereum protocols that rely on zero-knowledge proofs[5].

4.1 The Poseidon2 family of hash functions

Overview. Poseidon2 is built upon the $\mathsf{Poseidon2}^\pi$ permutation operating over $\mathbb{F}_p^t$ with $p > 2^{30}$ prime and $t \in \{2, 3, 4, 8, 12, 16, 20, 24\}$. The permutation is meant to be combined with either a compression function or a sponge construction to build a hash function. $\mathsf{Poseidon2}^\pi$ is based on the HADES design strategy which makes a distinction between external and internal rounds. Internal rounds (also called partial rounds) apply the nonlinear layer to only a part of the state, usually a single element, whereas external rounds (also called full rounds) process all elements in the same way. More precisely, $\mathsf{Poseidon2}^\pi$ processes an internal state $x = (x_0, \cdots, x_{t-1}) \in \mathbb{F}_p^t$ as follows:

$$\mathsf{Poseidon2}^\pi(x) = \mathcal{E}_{R_F-1} \circ \cdots \circ \mathcal{E}_{R_F/2} \circ \mathcal{I}_{R_P-1} \circ \cdots \circ \mathcal{I}_0 \circ \mathcal{E}_{R_F/2-1} \circ \cdots \circ \mathcal{E}_0(M_\mathcal{E} \cdot x)$$

where $\mathcal{E}$ and $\mathcal{I}$ refer to external and internal round functions iterated for R_F and R_P rounds, respectively. Note that a linear layer is applied before running the first external round, which differs from the original $\mathsf{Poseidon}^\pi$ design. The external/full round function is defined by:

$$\mathcal{E}(x) = M_\mathcal{E} \cdot \left(\left(x_0 + c_0^{(i)}\right)^d, \cdots, \left(x_{t-1} + c_{t-1}^{(i)}\right)^d \right)$$

where $d \geq 3$ is the smallest integer such that $\gcd(d, p-1) = 1$, $M_\mathcal{E}$ is a $t \times t$ maximum distance separable (MDS) matrix and $c_j^{(i)}$ is the j-th round constant for the i-th external round. The internal/partial round function is defined by:

$$\mathcal{I}(x) = M_\mathcal{I} \cdot \left(\left(x_0 + \hat{c}_0^{(i)}\right)^d, x_1, \cdots, x_{t-1} \right)$$

where $d \geq 3$ as before, $M_\mathcal{I}$ is a $t \times t$ MDS matrix and $\hat{c}_0^{(i)}$ is the round constant for the i-th internal round.

Efficient instantiations for hash-based signatures over MPC. Since Poseidon2 is a generic construction, all instantiations will most likely not provide the same level of MPC-friendliness. Because all operations but exponentiations can be computed locally in an MPC setting, one should aim at minimizing the d parameter as it would result in fewer multiplications. From a permutation-only perspective, one would also be tempted to minimize t and $R = R_F + R_P$ parameters as the amount of exponentiations is directly derived from them. However, at the hash function level, the optimal parameter selection depends on the input size to be processed. Indeed for large inputs that require a sponge mode as the underlying construction, having a large rate would allow to absorb more data per permutation, and eventually leading to fewer calls and fewer exponentiations in the end. In the case of hash-based signatures, most hash calls process small inputs to compute either hash chains from secret keys or nodes in the Merkle tree, with the exception of leafs which are obtained by hashing multiple public keys. This is why the generalized XMSS scheme from [?] instantiates Poseidon2 with the compression mode for chain and tree hashing, whereas it uses the sponge mode for leaf hashing. More specifically,

[5] https://www.poseidon-initiative.info/

their instantiations use a 31-bit prime field for efficient SNARK-based aggregation with $t = 16$ and $t = 24$ for chain and leaf hashing, respectively. We will hence focus on these parameters in the rest of this document.

4.2 Performance assessment

Hash chains over MPC. Benchmarking MPC protocols is challenging as their efficiency not only depends on the underlying cryptographic primitives but also on the security model, number of participants and network conditions. Because only nonlinear operations require communication between participants, the number of multiplications provides a good performance indicator. If a multiplication usually requires one communication round along with some precomputed data (*i.e.*, Beaver triples), in the case of exponentiation it is possible to lower the number of communication rounds at the cost of more precomputation. Therefore, the technique employed to compute x^d has a significant impact on performance. To estimate the number of communication rounds required for Poseidon2$^\pi$ over 31-bit prime fields, we used the MP-SPDZ framework [?][6] with the MASCOT protocol [?] which provides active security. The Poseidon2$^\pi$ implementation, meant to be compiled by MP-SPDZ, is available at https://github.com/ObolNetwork/pqdv/blob/main/poseidon2.mpc. We considered three different 31-bit prime fields which enable highly efficient implementation techniques, namely Mersenne31, KoalaBear and BabyBear. As reported in Table 2, even though the KoalaBear prime field leads to the highest number of rounds, it is actually more MPC-friendly[7] than the two others thanks to its minimal d value. Having now an idea on the number of communication rounds per permutation, it is necessary to evaluate how many permutation calls are needed to sign a message (it actually also matters for DKG, but we will assume it is not an issue at this stage as it suffers less timing constraints). To do so, we rely on the XMSS instantiations with Poseidon2 from [?] and use a script developped by the authors[8] to calculate the number of permutation calls to sign a message in the average case. According to the instantiations considered in Table 3, on average, the number of permutation calls for hash chains is at least 78. While at first glance it seems that $78 \cdot 30 = 2\,340$ communication rounds are required to generate a signature, a single communication round can actually be used for multiple multiplications as long as they do not depend on each other. In the case of hash-based signature, since each chunk is processed independently, it is possible to leverage these parallelization capabilities at the chunk level. Still, in the worst case where a w-bit chunk has value $2^w - 1$, then at least $(2^w - 1) \cdot 30$ rounds will be needed anyway due to the nature of hash chains. Like all hash-based signatures schemes, instantiations with low w parameter (typically $w \in \{1, 2\}$) are the most MPC-friendly as they require fewer hash calls. On the other hand, they lead to very a large signature size and will likely not be considered for deployment on the Ethereum beacon chain for this reason. Among the instantiations listed in Table 3, the ones with $w = 4$ seem the most promising as they provide a nice trade-off between signature size and running time. By leveraging parallelization capabilities, it would require $16 \cdot 30 = 480$ communication rounds to generate a signature, assuming KoalaBear prime field.

[6] https://github.com/data61/MP-SPDZ

[7] KoalaBear and BabyBear primes also show advantages over Mersenne31 when it comes to SNARKS thanks to their two-adic multiplactive subgroups for Cooley-Tukey NTTs.

[8] https://github.com/b-wagn/hashsig-parameters

Table 2: Poseidon2$^\pi$ parameters for 31-bit prime fields.

Prime p	**Parameters**				**Sbox impl.**	**MPC cost metrics**	
	t	d	R_F	R_P		(triples, squares)	com. rounds
$2^{31}-2^{24}+1$	16	3	8	20	x^3	$(148, 148)$	30
	24			23		$(215, 215)$	33
$2^{31}-1$	16	5	8	14	$(x^2)^2 \cdot x$	$(426, 0)$	66
	24			22		$(624, 0)$	90
$2^{31}-2^{27}+1$	16	7	8	13	$(x^2)^3 \cdot x$	$(423, 141)$	63
	24			21		$(639, 213)$	87

Table 3: Generalized XMSS instantiations with Poseidon2 over a 31-bit prime field. The reported number of permutation calls only considers hash chains during signature generation. For signature sizes, we consider two different leaf numbers, namely $L \in \{2^{18}, 2^{20}\}$. Regarding encodings, we refer to the original publication [**?**] for more details.

Encoding	**Parameters**		**Sig. size** (KiB)		**Perm. calls**
	w	chunks	$L=2^{18}$	$L=2^{20}$	(average case)
W	1	163	4.97	5.03	81
	2	82	2.75	2.81	123
	4	42	1.66	1.72	303
	8	22	1.11	1.34	2676
TSW ($\delta = 1$)	1	155	4.75	4.81	78
	2	78	2.65	2.7	117
	4	39	1.58	1.64	293
	8	20	1.06	1.27	2550

Precomputed hash chains. As mentioned in the previous section, another alternative would be to precompute all hash chain intermediate values, for each secret key and for every possible chunk. Therefore the memory usage would be $L \cdot (2^w - 1) \cdot \#\mathsf{chunks}$ digests where L refers to the number of leafs in the Merkle tree. For the instantiations reported in Table 3, where hashs are composed of either 7 or 8 field elements, it requires $\approx 2, 2, 4, 40$ GiB for $L = 2^{18}$ and $w = 1, 2, 4, 8$ respectively.

5 Future work

1. Benchmark with other protocols that are not natively supported by MP-SPDZ.
2. Investigate on the right security model. Malicious security seems a must but should we aim for (two-thirds?) honest majority or dishonest majority?

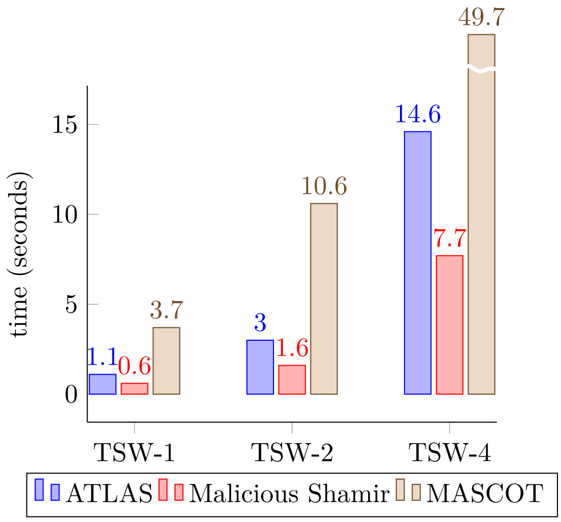

Figure 1: Benchmark results for hash chains calculations over MPC (online phase only) to sign a single message. Timing results are averaged over 10 runs in a network with 30ms delay.

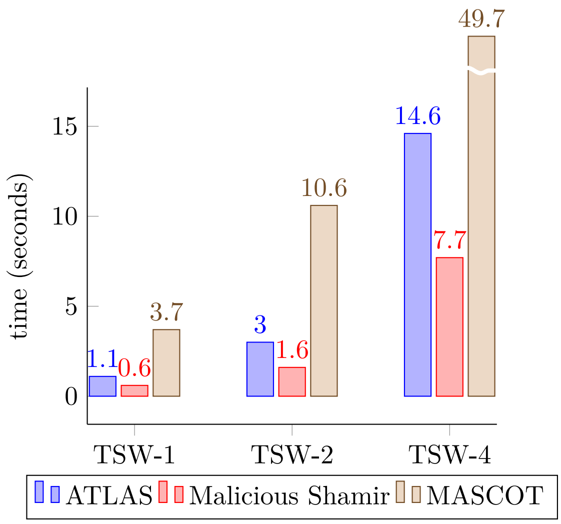

Figure 2: Benchmark results for hash chains calculations over MPC (offline phase only) to sign a single message. Timing results are averaged over 10 runs in a network with 30ms delay.