Scalable Batch Verification for Post-Quantum Hash-Based Signatures Using STARKs

Rodrigo Duarte de Meneses[1], **Marco Amaral Henriques**[1]

Faculdade de Engenharia Elétrica e de Computação
Universidade Estadual de Campinas – Campinas, SP – Brazil

r197962@dac.unicamp.br, maah@unicamp.br

Abstract. *This paper introduces a STARK-based batch verifier for a Merkle Signature Scheme (MSS) built from parallel Lamport-style one-time signatures and implemented in the Winterfell framework. The method compresses the validation of N signatures under a single Merkle root into a compact proof of 50-75 KiB. Verification then requires only a few dozen hash evaluations and completes in under 3.1 s for $N = 64$, outperforming naïve per-signature checks for all $N \geq 8$. Requiring no trusted setup, this approach paves the way for scalable, efficient validation of multiple post-quantum signatures.*

1. Introduction

As large-scale quantum computers threaten traditional digital signatures based on RSA or elliptic curves, hash-based signature schemes have emerged as a compelling post-quantum alternative. These algorithms rely only on the security of collision-resistant hash functions, avoiding complex hardness assumptions. Stateful schemes like LMS and XMSS have been standardized by NIST for limited use [NIST 2020]. These schemes use Merkle trees to aggregate many one-time signatures (e.g. Lamport or WOTS), providing strong security guarantees even against quantum attacks. However, each one-time keypair must be used exactly once, imposing careful state management and limiting practicality in high-volume or multi-party settings without additional infrastructure.

Hash-based signatures also impose large signature sizes and high verification costs, making per-signature checks infeasible in high-throughput settings such as Proof-of-Authority blockchain consensus [Wu et al. 2025]. To address this, succinct proof systems have been applied to batch-verify many one-time signatures in one compact proof, significantly reducing verifier workload [Chakraborty et al. 2025]. Khaburzaniya et al. introduced a STARK-based aggregator that compresses thousands of one-time signatures into a 100–200 KiB proof [Khaburzaniya et al. 2021], and Drake et al. extended STARK aggregation to hash-based multisignatures to replace BLS for Ethereum 2.0 [Drake et al. 2025]. Both rely on foundational STARK techniques for transparent, post-quantum proofs [Ben-Sasson et al. 2018b, Ben-Sasson et al. 2018a].

However, these STARK-based aggregation techniques typically require the verifier to receive every one-time public key, resulting in communication overhead proportional to the number of signatures. To address this issue, we present the first practical STARK-based batch verifier that efficiently integrates Lamport+ one-time signature checks with Merkle-tree authentication. In our approach, the verifier receives only a single Merkle root (ζ) for the entire batch and a concise Merkle proof (π), significantly reducing communication overhead. This technique yields succinct proofs (50–75 KiB) and maintains linear verification complexity.

2. Background

2.1. Hash-Based Digital Signatures

Hash-based signature schemes rely solely on the security of a collision-resistant hash function $H\colon \{0,1\}^* \to \{0,1\}^n$, and are therefore conjectured to be secure against quantum adversaries [NIST 2020]. Two important constructions for this work are Lamport+ one-time signature [Khaburzaniya et al. 2021] and its extension via a Merkle tree.

a) Lamport+ One-Time Signature. Lamport+ is an optimized variant of the classic bitwise Lamport OTS that halves key sizes by deriving each pk_i via a single hash of the corresponding sk_i [Khaburzaniya et al. 2021]. It is designed to be friendly to zero-knowledge and STARK-based proofs, since all operations reduce to simple hash evaluations and a final set-accumulator check. Public parameters include the message length m, a hash function H that maps arbitrary inputs to m bit outputs, and an accumulator interface (typically implemented as a Merkle tree) that compresses a set of n hash outputs into a single short public key.

LP.KeyGen: Sample n independent m-bit secret values $(sk_0, \ldots, sk_{n-1})$ uniformly. Each secret block is hashed to its public counterpart: $pk_i = H(sk_i)$. The private key is simply the list of secrets (sk_i), and the public key is either the full vector (pk_i) or, when key-size compression is required, the accumulator evaluated on $\{pk_i\}$.

LP.Sign: An m-bit message M is treated as an array of bits. For each bit i in $\{0, \ldots, n-1\}$, the signature element σ_i choice is: If $M_i = 1$, set $\sigma_i = sk_i$; if $M_i = 0$, set $\sigma_i = pk_i = H(sk_i)$. The full signature is the list (σ_i), with length n blocks, each m bits long.

LP.Verify: Given the message M, the signature σ, and the public key , each element x_i is computed as follows: if $M_i = 1$, set $x_i = H(\sigma_i)$; if $M_i = 0$, set $x_i = \sigma_i$. Finally, the verifier applies the accumulator to the set $\{x_i\}_{i=0}^{n-1}$ and checks that it matches the public key. If so, the signature is accepted; otherwise, it is rejected.

This construction requires a total of n hash invocations considering both signing and verification together (plus one accumulator evaluation). It's simplicity – limited to hashing and accumulator checks – makes Lamport+ particularly well-suited for concise algebraic representation required by STARK-based proof systems.

b) Merkle Signature Scheme. To sign multiple messages, one can use Lamport+ key pairs as the leaves of a Merkle tree. This yields a single short public key (the Merkle root), at the cost of appending a Merkle authentication path to each one-time signature:

Fix $N = 2^d$. Let $\big(sk^{(j)}, pk^{(j)}\big)$, $j = \{0, 1, \ldots, N-1\}$, be N independent Lamport+ key pairs, and define the leaf labels $L_j = H\big(pk^{(j)}\big) \in \{0,1\}^n, \quad j = 0, \ldots, N-1$. Build a binary Merkle tree of depth d by $T_\ell^{(k)} = H\big(T_{\ell+1}^{(2k)} \,\|\, T_{\ell+1}^{(2k+1)}\big), \ell = d-1, \ldots, 0; k = 0, \ldots, 2^\ell - 1$, with leaves $T_d^{(j)} = L_j$. The master public key is the root $\zeta = T_0^{(0)}$. To sign a message m with the jth key pair, one publishes $\sigma^{(j)} = \big(\mathsf{LP.Sign}(sk^{(j)}, m)\,,\ \mathsf{AuthPath}_j\big)$, where $\mathsf{AuthPath}_j$ is the sequence of sibling hashes along the path from L_j up to ζ. Verification recomputes the one-time signature check, then hashes up the tree and confirms the computed root equals ζ. Assuming H is collision-resistant and one-way, MSS is secure under one-time forking – each OTS key used only once guarantees that no leaf can be forged, and collision-resistance prevents Merkle path forgeries.

2.2. STARK Basics

Scalable Transparent ARguments of Knowledge (STARKs) are publicly verifiable proof systems that enable succinct validation of arbitrary computations without requiring a trusted setup [Ben-Sasson et al. 2018a]. By combining Interactive Oracle Proofs (IOPs) with cryptographic hash–based commitments, a STARK prover demonstrates the correctness of a computation in time roughly proportional to the computation's length multiplied by a polylogarithmic factor, while a verifier's workload grows only polylogarithmically in the computation size. The sole security assumption is the collision resistance of the underlying hash function, which endows STARKs with both transparency and post-quantum resilience. In contrast, alternative proof systems such as transparent SNARKs yield $O(\sqrt{N})$-size proofs and sublinear verification time, but still incur heavy prover costs (e.g., $O(N)$ in [Wei et al. 2025] or $O(N \log^2 N)$ in [Chen et al. 2023]). Empirical results confirm STARKs often outperform these systems in proof size and verification speed [Ernstberger et al. 2023, El-Hajj and Oude Roelink 2024].

The first essential component of a STARK is arithmetization, which translates a T-step computation into algebraic relations over polynomials. The prover first builds an execution trace – a table listing the program state at each time step – and then formulates polynomial constraints that hold exactly when each transition rule is obeyed. These constraints are compacted into a single "composition" polynomial defined over an extended domain, so that proving this polynomial has low degree is equivalent to showing every original constraint is satisfied across the trace. To verify the low-degree claim, STARKs employ the Fast Reed–Solomon IOP of Proximity (FRI), an IOP-based sub-protocol that uses Merkle-tree commitments and $O(\log N)$ oracle checks to achieve exponentially small soundness error [Ben-Sasson et al. 2018b]. Finally, a hash-based Fiat–Shamir transform collapses the interaction into a non-interactive proof, yielding a self-contained proof string with scalable verification and compact size.

3. STARK Construction

Our batch verification system is implemented as a STARK that proves the correctness of verifying N hash-based signatures (specifically, a MSS with Lamport+ OTS). We designed an Algebraic Intermediate Representation (AIR) that captures the entire verification procedure for a batch of signatures. In essence, the STARK's computation simulates an MSS verification algorithm over all N signatures in the batch, and the STARK proof attests that all signatures in the batch are valid with respect to a given Merkle root ζ.

To make the hashing within the signature verification amenable to arithmetization, we use the Rescue hash function [Sekulić et al. 2025] in place of standard cryptographic hashes. Rescue is a sponge-based hash that is designed to be efficient in arithmetic circuits (i.e., it is STARK-friendly). Figure 1 provides an overview of the architecture of our construction. The signers generate j Lamport+ OTS on a message M and publish their public keys as leaves in a Merkle tree whose root ζ becomes the master public key. The STARK prover takes as input message M, public keys pk, signature σ and root ζ, and produces a succinct proof π that (1) Lamport+ OTS on M is valid and (2) corresponding Merkle authentication path leads to ζ. The verifier validates the proof through the STARK verifier, where π attests that both the OTS and Merkle-tree constraints are satisfied.

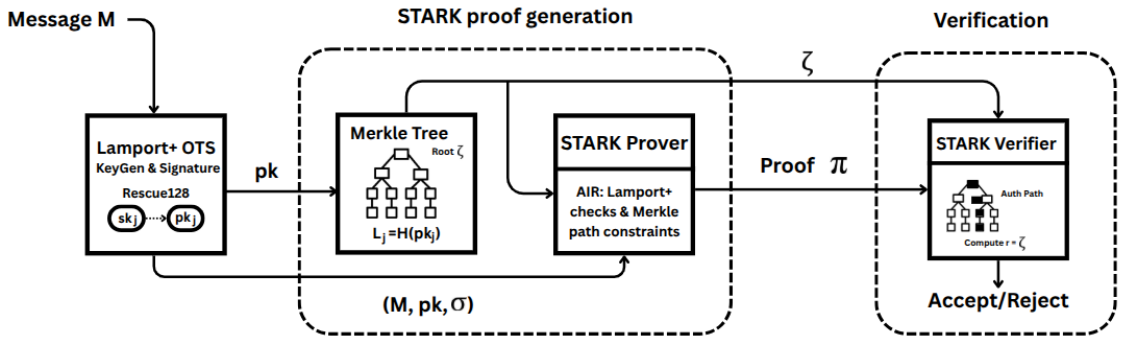

Figure 1. System overview of STARK-based batch verification of MSS signatures.

The use of an internal Merkle tree in the STARK proof generator offers several advantages compared to directly aggregating the Lamport+ signatures: Without the Merkle tree, the circuit would verify each of the N Lamport+ signatures individually, but would have to receive all of the public keys $\{pk_i\}$ as public inputs. By contrast, with the Merkle tree, the verifier only needs to receive π and ζ to check the entire set of signatures, dramatically reducing transmission overhead on the network.

4. Implementation & Experimental Setup

Our MSS batch-verification prototype is built on Winterfell[1] v0.12.2 by extending its built-in Lamport+ OTS and Merkle-path examples into a single unified STARK. We concatenate the signature-processing segment with the tree authentication segment into one execution cycle, using a periodic control bit to switch between them, and expanding trace registers and cycle lengths so that both batch size and tree depth can be adjusted without rewriting the core logic. During proof generation, the one-time signature and Merkle-path phases run in a single trace, with the control bit toggling which constraints apply at each step, leveraging Winterfell's existing hash-round implementation. The code targets Winterfell's public APIs, and employs the Rescue128 hash function for all cryptographic hashing inside the proof, as noted, to ensure the arithmetic constraints remain efficient.

For benchmarking, we measured the performance of key operations both with and without the STARK proof. We recorded the time to generate key pairs (KeyGen), the time to verify all signatures one-by-one without a proof (Verify), and the time to construct the Merkle tree for each batch size. On the STARK side, we measured the proof generation time and proof verification time, as well as the proof size in kibibytes. Data is available in https://github.com/regras/stark-mss. All experiments were conducted on an Intel Core i7-1185G7 CPU @ 3.00 GHz, running Ubuntu 22.04.4 LTS with 16 GB RAM.

5. Results

Table 1 reports the performance of our MSS-aggregate STARK implementation for batch sizes of $N = \{2, 4, 8, 16, 32, 64\}$, omitting the signing costs which were negligible.

Key-generation time, individual signature verification time, and Merkle-tree construction time all increase with complexity $O(N)$. In contrast, proof-generation time

[1] Available at https://github.com/facebook/winterfell.

Table 1. Individual signature verification compared to STARK aggregation (ms)

Batch Size	KeyGen	Verify	Merkle	Proof Gen.	Proof Verify	Proof Size (KiB)
2	306.9	207.2	0.2	3.6×10^4	68.0	46.7
4	604.2	405.7	1.1	1.4×10^5	142.5	51.9
8	1 214.2	794.2	2.3	5.8×10^5	323.9	55.9
16	2 427.5	1 600.2	6.2	2.3×10^6	662.8	62.6
32	4 812.0	3 158.4	12.6	9.2×10^6	1 430.7	68.4
64	9 711.1	6 292.9	24.8	3.6×10^7	3 058.0	75.2

increases superlinearly – approximately $O(N^2)$; therefore, doubling the batch size results in roughly four times more effort. Despite this heavy prover cost, proof sizes remain compact, growing only with $O(\log N)$, with an increase of about 6 KiB for each doubling of the batch. Notably, for $N \geq 8$, our approach provides smaller proofs than naively concatenating N Lamport+ signatures. Proof verification time also scales linearly with N, roughly doubling when the batch size doubles, but – even at $N = 64$ – verifying the STARK proof remains approximately twice as fast as verifying all signatures individually.

Table 2. Comparison of transparent SNARK and STARK constructions

	Prover Time	Proof Size	Verifier Time	Setup
[Wei et al. 2025] (PKC 2025)				
Log-space uniform circuits over Galois rings	$O(N)$	$O(\sqrt{N})$	$O(\sqrt{N})$	transparent
[Chen et al. 2023] (EUROCRYPT 2023)				
Arbitrary NP circuits over Galois rings	$O(N \log^2 N)$	$O(\sqrt{N})$	$O(\sqrt{N})$	transparent
Our STARK for MSS aggregation				
MSS with Lamport+ OTS	$O(N^2)$	$O(\log N)$	$O(N)$	transparent, post-quantum

Table 2 provides a comparison of asymptotics of our implementation with transparent SNARKs, highlighting trade-offs among prover time, verifier time, and proof size. Our STARK implementation delivers proof sizes of only $O(\log N)$ – markedly smaller than the $O(\sqrt{N})$ bound for transparent SNARKs – and introduces post-quantum security. These findings confirm that the STARK-based aggregation achieves substantial verifier-side savings and compact proof transmission at the expense of increased prover workload.

6. Conclusion

We presented a practical STARK-based batch verification scheme for post-quantum hash-based signatures, focusing on a Merkle-tree-based Lamport+ signature system. By merging the one-time signatures and Merkle path checks into a single proof, a verifier can validate N signatures using only a few dozen hash operations and a single proof check. The trade-off is a longer proving time, which can be amortized in scenarios where many signatures must be verified and a single party is willing to incur the proving cost.

Like all Lamport-style constructions, our scheme remains stateful (each key pair must be tracked as it can only be used once), which requires careful state management to ensure that OTS keypairs are not reused. Eliminating the stateful requirement and improving prover performance are necessary steps toward a fully general batch-verification framework. We also plan to evaluate the security and throughput of our batch-verification proof in a production setting (e.g. multiple signatures in blockchain consensus schemes).

References

Ben-Sasson, E., Bentov, I., Horesh, Y., and Riabzev, M. (2018a). Fast Reed-Solomon Interactive Oracle Proofs of Proximity. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107, pages 14:1–14:17, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Ben-Sasson, E., Bentov, I., Horesh, Y., and Riabzev, M. (2018b). Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046.

Chakraborty, S., Hofheinz, D., Langrehr, R., Nielsen, J. B., Striecks, C., and Venturi, D. (2025). Malleable snarks and their applications. In *Advances in Cryptology – EUROCRYPT 2025*, pages 184–213, Cham. Springer Nature Switzerland.

Chen, B., Bünz, B., Boneh, D., and Zhang, Z. (2023). Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In *Advances in Cryptology – EUROCRYPT 2023*, Lecture Notes in Computer Science, pages 499–530. Springer.

Drake, J., Khovratovich, D., Kudinov, M., and Wagner, B. (2025). Hash-based multi-signatures for post-quantum ethereum. Cryptology ePrint Archive, Paper 2025/055.

El-Hajj, M. and Oude Roelink, B. (2024). Evaluating the efficiency of zk-snark, zk-stark, and bulletproof in real-world scenarios: A benchmark study. *Information (Switzerland)*, 15(8). Publisher Copyright: © 2024 by the authors.

Ernstberger, J., Chaliasos, S., Kadianakis, G., Steinhorst, S., Jovanovic, P., Gervais, A., Livshits, B., and Orrù, M. (2023). zk-bench: A toolset for comparative evaluation and performance benchmarking of SNARKs. Cryptology ePrint Archive, Paper 2023/1503.

Khaburzaniya, I., Chalkias, K., Lewi, K., and Malvai, H. (2021). Aggregating and thresholdizing hash-based signatures using STARKs. Cryptology ePrint Archive, Paper 2021/1048.

NIST (2020). Recommendation for Stateful Hash-Based Signature Schemes. NIST Special Publication 800-208, National Institute of Standards and Technology.

Sekulić, J., Čapko, D., Erdeljan, A., Grbić, T., and Nenadić, K. (2025). A short survey of zk-friendly hash functions. In *2025 24th International Symposium*, pages 1–5.

Wei, Y., Zhang, X., and Deng, Y. (2025). Transparent snarks over galois rings. In Jager, T. and Pan, J., editors, *Public-Key Cryptography – PKC 2025*, pages 418–451, Cham. Springer Nature Switzerland.

Wu, F., Zhou, B., Song, J., and Xie, L. (2025). Quantum-resistant blockchain and performance analysis. *The Journal of Supercomputing*, 81(3).