Looma: A Low-Latency PQTLS Authentication Architecture for Cloud Applications

Xinshu Ma and Michio Honda
University of Edinburgh

***Abstract*—Quantum computers threaten to break the cryptographic foundations of TLS, prompting a shift to post-quantum cryptography (PQC). However, PQ authentication imposes significant performance overheads, particularly for mutual TLS (mTLS) in cloud environments where handshake rates are high. We present Looma, a fast PQ authentication architecture that splits authentication into a fast, on-path sign/verify operation and slow, off-path precomputations performed asynchronously, reducing handshake latency without sacrificing security. Integrated into TLS 1.3, Looma lowers PQTLS handshake latency by up to 44% compared to Dilithium-2. Our results demonstrate practicality of Looma for scaling secure communications in the post-quantum era.**

I. Introduction

Quantum computers are no longer a mere theoretical possibility. Recent advances from Google, IBM, and Oxford [1], [2], [3], [4] have demonstrated steady progress toward scalable, error-corrected quantum hardware, making large quantum machines increasingly practical. Meanwhile, classical cryptography relies on mathematical problems, such as factoring large numbers, that are considered hard for conventional computers but are efficiently solvable by quantum computers. For instance, a recent study from Google shows that a quantum computer with a million logical qubits could break RSA-2048—the most widely used digital signature algorithm—in less than a week [5]. The same threat extends to other widely deployed schemes like ECDSA and ECDHE, which underpin authentication and key exchange in TLS.

This impending risk has motivated the global shift toward post-quantum cryptography (PQC), a family of algorithms designed to withstand quantum attacks. As TLS is the backbone of secure communications across the Internet and in the cloud, there is an urgent need to transition from classic TLS to Post-Quantum TLS (PQTLS). Unfortunately, PQC schemes generally impose much higher computational and bandwidth costs compared to classical algorithms, posing a significant barrier to deployment—especially at cloud scale.

In modern cloud environments, the move from monolithic architectures to microservices and serverless functions has further exacerbated these challenges. Each microservice operates independently, with frequent connections triggering a new TLS handshake per interaction. This architecture dramatically increases the volume of cryptographic operations. At the same time, cloud operators employ mutual authentication (mTLS), moving beyond the server-only authentication that dominates the public Internet. The combined effect is a dramatic increase in handshake-related cryptographic cost and, hence, connection latency—a bottleneck further amplified by the overheads of post-quantum schemes.

Adopting PQC in datacenters thus demands practical, high-performance solutions. While industry leaders such as Amazon [6], Cloudflare [7], [8], and Google [9], [10], [11] have made important progress toward efficient PQTLS—primarily by optimizing key exchange—authentication remains a performance bottleneck, particularly in mTLS deployments.

Motivated by this gap, we introduce Looma[1], a general framework for cloud applications that significantly accelerates PQTLS handshakes by integrating the online/offline signature paradigm into the TLS workflow. Looma replaces the on-path post-quantum signature with an online/offline variant, splitting authentication into two parts: (1) a fast, on-path sign/verify operation executed during the handshake, and (2) slow, off-path precomputations performed asynchronously. A dedicated KeyDist service manages the distribution of precomputed keys. Moreover, Looma is signature-agnostic, making it compatible with various signature-based TLS authentication mechanisms. Compared to PQTLS with Dilithium-2, Looma reduces TLS handshake latency by up to 37 % at P50 and 42 % at P99 with server-only authentication, and up to 44 % at P50 and 42 % at P99 with mutual authentication.

Our contributions are the following:

1) An in-depth analysis of post-quantum mTLS handshake latency, demonstrating the need for accelerated authentication.
2) The design of Looma, a fast PQTLS authentication architecture that decouples expensive authentication from the latency-critical handshake path and allows safe fallback to alternative signature schemes.
3) A parameter analysis of Looma, identifying configurations suitable for common deployment scenarios.
4) An implementation of Looma, fully integrated into TLS 1.3 by extending the Picotls library, with support for any Picotls-based application or protocol.
5) An extensive evaluation showing that Looma consistently outperforms PQ signature algorithms, even classic ones,

Network and Distributed System Security (NDSS) Symposium 2026
23 - 27 February 2026 , San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.[23|24]xxxx
www.ndss-symposium.org

[1]Low-latency Online/Offline Mutual Authentication

particularly for mutual authentication in PQTLS.

II. Motivation

A. Why Fast (PQ)TLS Handshake Matters in the Cloud?

TLS handshake performance is critical in modern cloud environments for several reasons:

First, **microservice architectures**—prevalent in cloud platforms powering social media, ride-sharing, and privacy-preserving analytics [12], [13], [14], [15]—break applications into many small, frequently interacting services. Each network connection typically requires a new TLS handshake, and the ephemeral, autoscaled nature of containers and pods [16] leads to extremely high handshake rates. Because instances are frequently created, destroyed, and rescheduled, techniques like TLS session resumption or pooling are often ineffective.

Second, **service mesh deployments** (e.g., Istio) and sidecar proxies (e.g., Envoy, Linkerd, ServiceRouter [17]) introduce even more handshakes per request, as traffic is transparently encrypted and routed through multiple hops for security and observability.

Third, latency-sensitive, **short-lived flows** dominate inter-service communication in the cloud [18], [19]. Many requests complete in less time than it takes to perform a TLS handshake, making handshake latency a substantial portion of end-to-end service time.

Finally, modern datacenter networks are engineered for **ultra-low fabric latency** (10-50 μs) and minimal buffering [20], shifting bottlenecks to the hosts themselves—where cryptographic operations, especially PQ handshakes, can dominate application delay and resource usage.

In summary, the scale, churn, and short-flow patterns of cloud-native systems have transformed handshake overhead from a negligible concern on the Internet into a core performance bottleneck. Fast (PQ)TLS handshake is now essential for efficient, responsive cloud services.

B. Existing TLS Handshake Accelerations

The optimization of TLS handshake performance—particularly for cryptographic operations—has garnered significant research attention. Existing approaches fall into three primary categories.

a) Offloading: Offloading classic asymmetric cryptographic operations to specialized hardware can improve handshake performance, but faces notable constraints. Solutions such as SSLShader [21] leverage GPUs, while Kim et al. [22] propose offloading TLS handshakes to SmartNICs, albeit with limitations imposed by device processing power. Canal Mesh [23], deployed in Alibaba Cloud, employs a centralized key server to handle asymmetric operations for mTLS. While offloading can boost handshake *throughput* by freeing host CPU cycles, it does little to reduce handshake *latency*, as data transfer over the PCIe bus or network adds overhead. Additionally, these approaches require trusting the offload device and heavily depends on hardware capabilities, which reduces general applicability.

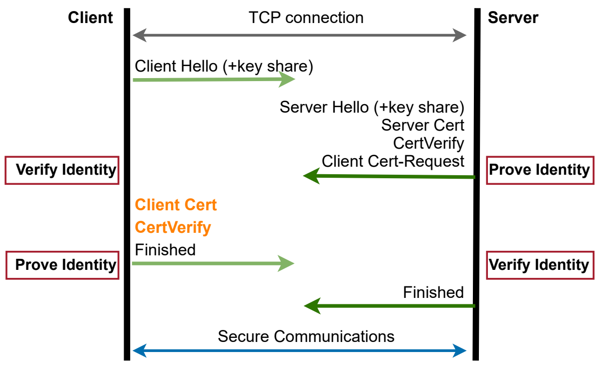


Fig. 1: Mutual authentication in mTLS Handshake.

b) Network Protocol Enhancement: Protocol-level modifications can reduce the number of network round-trips required for TLS handshakes. Examples include TLS 1.3's 1-RTT handshake [24] and session resumption [25], both of which simplify connection setup to lower latency. Approaches such as ZTLS achieve zero-RTT handshakes by pre-distributing certificates and public key shares via DNS, though this requires stronger trust assumptions [26]. Post-quantum proposals like KEMTLS replace signature-based authentication with key encapsulation mechanisms as KEM is slightly faster than PQ signature algorithms, achieving quantum-safe mutual authentication with reduced bandwidth at the cost of an additional RTT [27]. However, as highlighted in § II-A, the main bottleneck in modern datacenters is not network latency (i.e., rrt time)—which is already very low (on the order of tens of microseconds)—but the computational cost of cryptographic operations. Thus, protocol enhancements alone are insufficient to address handshake latency in the cloud.

c) Cryptographic Operation Optimization: Substantial performance gains are possible by optimizing cryptographic primitives at both the mathematical and implementation level. Recent advances have accelerated classical algorithms such as X25519 and Ed25519 using techniques like the Montgomery ladder combined with AVX-512 vectorization [28]. In the post-quantum setting, highly optimized implementations of SNTRUP761 have significantly improved key exchange performance in OpenSSL [29]. Notably, most of these works focus on accelerating key exchange.

Orthogonal to these approaches, our work (Looma) targets handshake acceleration by decoupling expensive post-quantum authentication from the latency-critical path, achieving significant reductions in handshake latency while preserving protocol compatibility.

C. What We Focus: PQ Authentication and Costs

a) TLS and mTLS: TLS is the standard protocol for end-to-end secure channels in both Internet and cloud environments. While typical Internet connections only authenticate the server, datacenter and cloud deployments often require mutual authentication (mTLS), where both endpoints cryptographically verify each other's identities. mTLS is essential

TABLE I: Client-side and Server-side TLS and mTLS handshake components.

Handshake	Keygen+ex	Verify cert	Sign	Verify
TLS$_c$	✓	✓	✗	✓
TLS$_s$	✓	✗	✓	✗
mTLS	✓	✓	✓	✓

TABLE II: Overheads of asymmetric cryptographic operations in TLS handshake.

Operation	Algorithm	Latency (μs)	PQ	Lib
Keygen+ex	ECDHE	45.20	✗	OpenSSL
	Kyber-512	74.70	✓	OQS
Verify cert	RSA-2048	17.22	✗	OpenSSL
	Dilithium-2	26.27	✓	OQS
	Falcon-512	30.7	✓	OQS
Sign	ECDSA	14.92	✗	OpenSSL
	RSA-2048	200.37	✗	OpenSSL
	Dilithium-2	51.74	✓	OQS
	Falcon-512	149.2	✓	OQS
Verify	ECDSA	40.18	✗	OpenSSL
	RSA-2048	15.32	✗	OpenSSL
	Dilithium-2	21.27	✓	OQS
	Falcon-512	28.3	✓	OQS

for enforcing strict access controls, preventing man-in-the-middle attacks during key exchange, and ensuring connections to trusted endpoints—critical for sensitive or regulated cloud services.

The mTLS handshake, illustrated in Figure 1, uses X.509 certificates and digital signatures for authentication. In this process, both client and server present certificates signed by a trusted Certificate Authority (CA) and use digital signatures (`CertVerify`) to prove possession of their private keys over the handshake transcript. The peer verifies these signatures using the public key embedded in the certificate and the CA's public key from its trust store.

Compared to standard TLS (server-only authentication, denoted as sTLS), mTLS adds two extra handshake messages (see orange highlights in Figure 1) and requires additional asymmetric operations. The client must sign a `CertVerify` message, and the server must verify both the client's certificate and signature (Table I). While certificate messages themselves incur no runtime cryptographic cost (as certificates are pre-issued), mTLS increases the number of expensive sign and verify operations during the handshake (see Table I), amplifying the computational burden—especially for post-quantum algorithms.

b) Costs: While mutual authentication enhances security, it introduces measurable computational overhead. Cryptographic operations encompass the four primary cryptographic actions outlined in Table II, as well as all message encryption and decryption (i.e., symmetric crypto) during the handshake. Note that our measurement was performed locally without network connection to highlight the trade-off between enhanced security in mTLS and its associated computational overheads.

Further latency analysis of asymmetric cryptographic operations in Table II underscores the significant performance impact of authentication-related processing during the TLS handshake. The table compares both post-quantum (PQ) and classical cryptographic schemes used for key exchange and digital signatures. Since key exchange and certificate verification occur in both TLS and mTLS, our primary focus is on the additional *Sign* operation introduced at the client side during mTLS handshake (as highlighted in Table I).

For example, in an mTLS client, the fastest PQ signature scheme, Dilithium-2, incurs 51.7μs for signing compared to 14.9μs for the classical ECDSA scheme. RSA exhibits the highest mutual authentication cost, with signing requiring 200μs–significantly higher computational cost despite its fast verification. The total mTLS handshake time closely aligns with the latency of the *Sign* operation, further emphasizing its performance impact (see Table II).

These results suggest that ECDSA's lower *Sign* latency positions it as a more efficient choice for mTLS implementations. In contrast, the higher signing costs associated with Dilithium-2 and Falcon-512 contribute to increased total handshake latency, reinforcing the performance trade-offs inherent to current PQ signature schemes. This disparity becomes critical in datacenter environments dominated by short-lived connections and high concurrency, where cryptographic overhead can constitute up to the majority of total handshake latency, directly impacting application-level performance through increased connection establishment times.

In sum, the transition to PQ security amplifies the performance challenges of mTLS. While PQ verification can outperform classical algorithms (e.g., Dilithium-2 verification is 47% faster than ECDSA), PQ signing operations exhibit $3-5\times$ higher latency than their classical counterparts—a critical bottleneck in mTLS handshakes dominated by signing workloads. This performance-security trade-off necessitates new approaches to maintain the low-latency requirements of modern datacenter infrastructure while ensuring quantum resistance.

III. Preliminaries

A. Digital Signatures

Digital signatures are fundamental cryptographic primitives that ensure the authenticity and integrity of digital messages. A signature scheme is a tuple of algorithms $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ together with an associated message m.

$(pk, sk) \leftarrow \mathsf{KeyGen}(\kappa)$: Given the security parameter κ, the key-generation algorithm outputs a private signing key sk, and a public verification key pk.

$\sigma \leftarrow \mathsf{Sign}(m, sk)$: On input a message m, the signing algorithm outputs a signature σ under sk associated with m.

$0/1 \leftarrow \mathsf{Verify}(m, \sigma, pk)$: On a message–signature pair (m, σ), the verification algorithm outputs a bit b under pk, where $b = 1$ signifies *accept* and $b = 0$ signifies *reject*.

Unlike classical signature schemes such as RSA and ECDSA—which succumb to Shor's algorithm on a sufficiently powerful quantum computer—PQ signatures derive their security from mathematical problems believed to remain difficult even for quantum adversaries. Among the various PQ families, the National Institute of Standards and Technology (NIST) has selected two primary classes for standardization: lattice-based and hash-based signatures. Its three finalist algorithms are Dilithium (lattice-based) [30], Falcon (lattice-based) [31], and SPHINCS$^+$ (hash-based) [32]. These schemes now define the baseline for quantum-resilient authentication.

B. Online/offline Signature Paradigm

The online/offline signature paradigm, pioneered by Even, Goldreich, and Micali (EGM) [33], [34]—transforms any conventional, computation-heavy signature scheme into a two-phase process: 1) *offline* (or pre-computation) phase that executes heavy computations, independent of any message to be signed, and 2) *online* phase that is typically very fast when the message is ready. It uses a one-time signature scheme, a scheme that enjoys high security but can sign only one message, as a building block. The essence is to apply (offline) the ordinary signing algorithm to authenticates a lightweight one-time public key, and then to apply (online) the one-time signing algorithm, which incurs only negligible latency. This division retains the robustness of the ordinary signature scheme while reducing per-message latency to a negligible level.

Formally, an online/offline signature scheme is a tuple of algorithms $(\mathsf{KeyGen}, \mathsf{PreSign}, \mathsf{FastSign}, \mathsf{Verify})$ with a message m.

$(pk, sk) \leftarrow \mathsf{KeyGen}(\kappa)$**:** Given the security parameter κ, the key-generation algorithm outputs a private signing key sk, and a public verification key pk.

$\rho \leftarrow \mathsf{PreSign}(sk)$**:** On input a signing key sk, the presigning algorithms outputs a pre-signing state ρ.

$\sigma \leftarrow \mathsf{Sign}(m, \rho, sk)$**:** On input a message m, the signing algorithm outputs a signature σ under sk and associated with ρ.

$0/1 \leftarrow \mathsf{Verify}(m, \sigma, pk)$**:** On a message–signature pair (m, σ), the verification algorithm outputs a bit b under pk, where $b = 1$ signifies *accept* and $b = 0$ signifies *reject*.

C. Winternitz One-Time Signature Plus

The WOTS$^+$ hash-based signature scheme—formalised by Hülsing et al.,[35]—sharpens the classical Winternitz approach by tightening security proofs and shrinking both signature and public-key sizes relative to earlier one-time variants. Thanks to this efficiency–security balance, WOTS$^+$ has been adopted as the leaf-level signing algorithm in two flagship hash-based, post-quantum schemes: SPHINCS$^+$ [36] and XMSS (eXtended Merkle Signature Scheme) [37]. XMSS, standardised by the IETF in 2018, is regarded as the most mature member of the hash-based signature family, and its deployment experience further attests to the practicality of WOTS$^+$ as a foundational component.

Figure 2 illustrates the core idea of the WOTS$^+$ signature scheme, which relies on applying a fixed number of iterations of a chaining function (denoted by c, typically instantiated as a cryptographic hash function) starting from

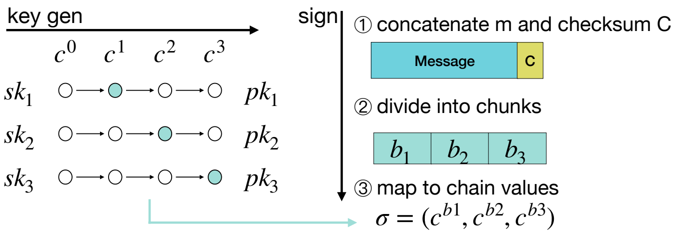


Fig. 2: A toy example of WOTS$^+$ signature scheme: key generation and signing.

random inputs. These random values form the secret key: $SK = (sk_1, sk_2, sk_3)$. The corresponding public key $PK = (pk_1, pk_2, pk_3)$ is derived by applying the chaining function $w - 1$ times to each secret key element.

To sign a message m, WOTS$^+$ first encodes the message and its checksum in base-w (with $w = 4$ in this example), producing a sequence of integers that is then split into a certain number of blocks. Each block value determines how many times the chaining function is applied to the corresponding secret key component. In the illustrated example, the base-w encoding yields three chunks $(b_1, b_2, b_3) = (1, 2, 3)$, and the signature is computed by applying the chaining function b_i times to sk_i. Thus, the resulting signature is constructed as $\sigma = (c^{b_1}(sk_1),\ c^{b_2}(sk_2),\ c^{b_3}(sk_3))$. To verify a signature, each element of σ is hashed the required of number of times to get to $w - 1$.

IV. Design Space

A. TLS Ecosystem in the Cloud

The security and communication landscape within datacenters—particularly for microservices and serverless functions—differs markedly from that of the broader Internet. Two characteristics are especially relevant for our approach:

a) Narrow, Stable Peer Sets: Unlike the Internet, where endpoints can interact with a vast number of unknown parties, datacenter services typically communicate with a small, stable set of peers. Production studies show that, for each user request, Alibaba's large-scale microservices backend has a median fan-out of just 2 and rarely exceeds 10 downstream calls. Meta's social-media stack is slightly broader (median 6–8), but over 95% of requests still reach fewer than 20 services [38]. In serverless environments, most workflows involve 1–5 functions and typically interact with only a handful of managed services (such as storage, databases, or queues) [39]. This fixed and limited peer set enables opportunities for optimizations such as pre-computing cryptographic material or caching verification keys to reduce authentication overhead.

b) Separation of Network and Application Logic: A defining feature of modern microservice and serverless architectures is the decoupling of network functions—such as TLS termination, certificate management, and traffic monitoring—from application logic. These tasks are routinely handled by infrastructure-level proxies or service mesh components, which operate independently from application code.

Our design builds on this architectural separation: by shifting expensive authentication operations off the latency-critical path and into background tasks managed by independent service, we enable efficient key dissemination and asynchronous cryptographic preparation. This loosely coupled yet well-defined structure is particularly well-suited for cloud environments, allowing secure, scalable mutual authentication without impeding application performance or developer agility.

Threat model. We assume that standard cryptographic primitives remain sound: authenticated-encryption schemes provide confidentiality and integrity, collision-resistant hash functions and existentially unforgeable digital-signature schemes cannot be broken by the adversary.

Any individual micro-service may be taken over through remote-code-execution flaws, container escape, or insider misuse. A compromised instance can then forge or misuse credentials, impersonate peer services, and exfiltrate or tamper with sensitive data. With a single foothold the attacker effectively gains power [40]: they can observe, intercept, replay, delay, modify, or inject packets, enabling eavesdropping and man-in-the-middle attacks during session establishment. We trust the underlying cloud substrate (hardware, firmware, hypervisor, host OS) and assume that data-plane helpers (i.e., sidecar proxies) and control-plane services — service discovery, the internal CA, orchestrators, policy engines — start life correctly configured and uncompromised. Yet, following a zero-trust philosophy, we do not grant them perpetual trust: any such component might later be subverted via exploits, misconfiguration, insider abuse, or supply-chain attacks.

Scope. We focus exclusively on TLS-secured communications within the same datacenter and explicitly exclude connections between external clients and datacenter applications or gateways. Such intra-datacenter networks typically exhibit extremely low latency ($\approx$20 μs) and high bandwidth (e.g., 100Gbps). To meet the demands of latency-sensitive workloads, our proposed Looma scheme must be optimized specifically for performance and scalability within this context. Rather than pursuing a general-purpose authentication solution applicable to diverse settings (such as the open Internet, or mobile networks), we target designs that deliver an optimal trade-off tailored to modern datacenters. Furthermore, we aim to provide an industry-standard security level (128-bit security) while significantly improving application-level performance.

Other secure communication schemes that operate at different layers or target distinct deployment models are orthogonal. IPsec [41] functions at the network layer, establishing operator-managed host-to-host tunnels within the trust domain. TLS with pre-shared keys (PSK) [42], used for session resumption, reduces cryptographic overhead but sacrifices forward secrecy and does not apply to full handshakes. QUIC [43] builds upon the TLS 1.3 handshake at the transport layer.

B. Design Choice

While online/offline signatures are well-studied in theory, real-world deployments have largely focused on constrained embedded devices. Looma demonstrates how this paradigm can dramatically reduce PQTLS handshake latency and boost aggregate throughput in high-performance datacenter environments.

Among the classical frameworks for converting a standard signature scheme into an online/offline variant—namely, the Even–Goldreich–Micali (EGM) construction [33] using one-time signatures, and the Shamir–Tauman (ST) approach [44] relying on trapdoor hashes (Table VIII)—we deliberately choose the *hash-only* EGM pathway. This choice leverages cryptographic hash functions already trusted and widely used in TLS, introducing no new cryptographic assumptions or compliance requirements. In contrast, ST-style variants, such as those combining Falcon-512 with lattice-based trapdoor hashes [45], though potentially offering shorter signatures, depend on newer and less-vetted primitives.

As a concrete realization, we wrap Dilithium-2 with a WOTS$^+$ one-time signature layer. We select Dilithium-2 as it is currently the fastest PQ signature scheme and NIST's recommended default; both Dilithium-2 and WOTS$^+$ are part of the NIST post-quantum portfolio and have undergone extensive public scrutiny. The resulting Dilithium-2/WOTS$^+$ combination maintains purely hash-based post-quantum security and, once verification keys are pre-distributed, incurs only a modest size overhead—making it a practical and defensible first step toward online/offline authentication in datacenter TLS. Note that the latency improvement advantages cannot be delivered during TLS session resumption because authentication is omitted there.

V. Looma Design

We present the Looma high-level architecture (§ V-A), emphasizing its low-latency authentication path (see § V-B), which accelerates request handling by partitioning the otherwise expensive signing operation. We then detail the background key-provisioning mechanism (§ V-D), responsible for continuously supplying fresh cryptographic material to the foreground plane.

A. Looma Architecture

Figure 3 illustrates the integration of the Looma architecture into the mTLS handshake. The handshake begins with the client sending a `ClientHello` message that advertises support for the Looma signature algorithm. In response, the server transmits a `ServerHello`, optional encrypted extensions, its certificate, and a `CertVerify` message containing a lightweight Looma signature computed over the handshake transcript. The client similarly follows up by presenting its own certificate and corresponding lightweight Looma signature in a `CertVerify` message. Both endpoints subsequently exchange `Finished` messages and transition to encrypted application data.

To minimize handshake latency, Looma shifts computationally expensive PQ signature operations away from the critical path. Internally, each endpoint is organized into two logical planes: a *foreground plane*, which performs latency-critical

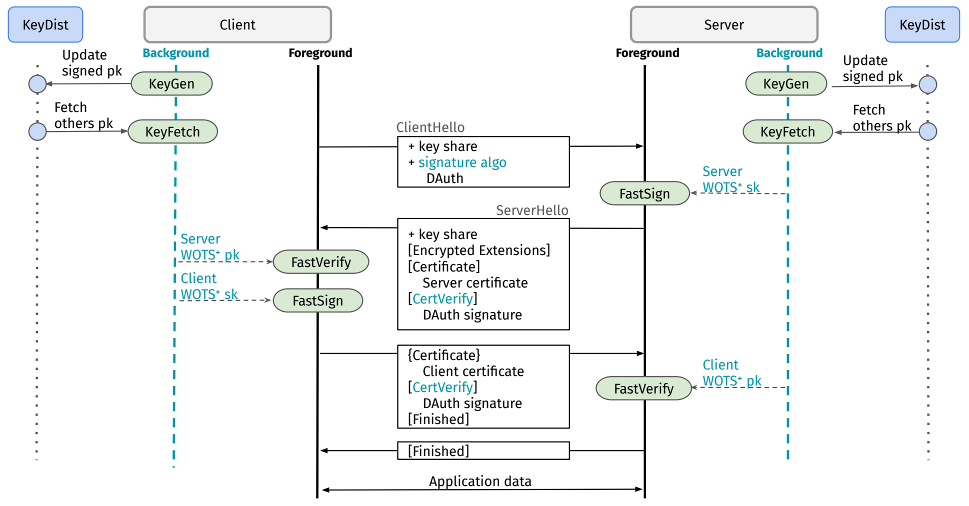


Fig. 3: mTLS 1.3 handshake process with Looma framework for authentication. Dashed green box/text indicates Looma integration.

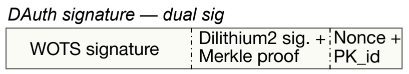
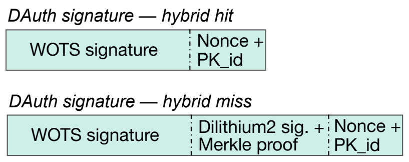


Fig. 4: Layout of Looma signatures when using dual-sig mode.



DAuth signature — hybrid hit
WOTS signature
Nonce + PK_id
DAuth signature — hybrid miss
WOTS signature
Dilithium2 sig. + Merkle proof
Nonce + PK_id

Fig. 5: Layout of Looma signatures when using hybrid mode.

operations such as rapidly issuing one-time WOTS⁺ signatures (FastSign) and quickly authenticating peer signatures (FastVerify), and a *background plane*, responsible for proactive key management. The background plane operates two maintenance tasks concurrently: KeyGen, which continuously generates local WOTS⁺ secret keys and uploads corresponding signed public keys, and KeyFetch, which regularly retrieves updated public keys from peers via the *KeyDist* service.

KeyDist is a storage-based service tailored for distributing batches of pre-generated and signed cryptographic public keys within cloud or datacenter microservice environments. By design, it serves solely as a repository for freshly signed public keys by each endpoint, relying on cryptographic verification rather than KeyDist-side trust. This separation of concerns allows Looma to achieve smooth, low-latency handshakes while providing scalable and secure public-key distribution.

Each service endpoint is provisioned with a Dilithium-2 key pair $(PK_{\mathrm{d2}}, SK_{\mathrm{d2}})$. The public key PK_{d2} is submitted to the internal CA to obtain an X.509 certificate. In Looma, we accelerate the mTLS handshake by replacing the computationally expensive Dilithium-2 signing and verification operations with a lightweight online/offline signature paradigm, preserving post-quantum security while sharply reducing authentication latency.

B. Foreground Plane: Fast Authentication

The foreground plane provides a synchronous interface optimized for rapid signature generation and verification over the handshake transcript (HT), thus ensuring minimal latency during handshake processing.

1) Signing Operation: To achieve fast, on-demand authentication, the foreground plane relies on a pool of pre-generated WOTS⁺ key pairs prepared by the background plane. Consequently, secret keys (SK) and the corresponding peer's public keys are readily available in their respective queues prior to the handshake initiation.

When generating a `CertVerify` message, the signer dequeues a fresh WOTS⁺ secret key (SK) from its local key queue and computes a lightweight WOTS⁺ signature:

$$\sigma \leftarrow \mathsf{FastSign}(HT, SK, r)$$

where HT is the fixed-length handshake transcript, and r is a nonce associated with the signature. The resulting signature σ is then embedded into the outgoing handshake message.

2) Verification Operation: Upon receiving a peer's `CertVerify`, the verifier employs a pre-cached WOTS⁺ public key (PK) to efficiently verify the authenticity of the incoming signature:

$$0/1 \leftarrow \mathsf{FastVerify}(HT, \sigma, PK, r)$$

Specifically, the verification process leverages the properties of the WOTS⁺ scheme by reconstructing an expected public key PK^* from the received signature σ, the handshake transcript HT, and the nonce r. The verifier then compares this derived public key PK^* with the pre-cached public key PK. The algorithm outputs a binary decision bit b, where $b = 1$ indicates a successful match (acceptance), and $b = 0$ indicates a mismatch (rejection).

C. Fallback Strategy

We now consider the important fallback scenario, specifically addressing the cache-miss problem where a verifier does not have the corresponding public key pre-fetched prior to the TLS handshake. Cache misses typically occur during the first handshake between a client and server or when the server fails to refresh a client's public keys in time. Note that cache misses are rare on the client side, as the client initiates connections and thus ensures the server's public key is pre-fetched before handshake initiation.

We propose two distinct solutions to handle such cache-miss scenarios, each with clear trade-offs.

Option 1: Dual Signature Mode. As illustrated in Figure 4, each WOTS$^+$ signature is accompanied by an additional Dilithium2 signature (signed over the Merkle tree root) along with a Merkle tree inclusion proof verifying the individual WOTS$^+$ public key. During verification, the server first checks its cache for a matching pre-verified public key. If present, the verifier proceeds directly with the efficient FastVerify. Otherwise, it utilizes the included Dilithium2 signature (verified with the PK_{d2} extracted from the `Certificate`) and the Merkle tree inclusion proof to validate the derived WOTS$^+$ public key (PK^*).

This dual-signature strategy ensures seamless authentication even with a 0% public key cache hit rate. However, it incurs a constant overhead of approximately 2044 bytes (Dilithium2 signature size) plus the Merkle inclusion proof size, which is $\log_2 B \times 32$ bytes (given a binary Merkle tree with B leaves and 32-byte hashes). For instance, a Merkle tree with 1024 leaves produces a proof size of 320 bytes. This overhead persists even in scenarios with high cache-hit rates, potentially increasing unnecessary bandwidth usage.

Option 2: Bloom Filter Hybrid Mode. To address the overhead inherent in the dual signature mode, we introduce a more dynamic hybrid strategy. In this mode, each server maintains a Bloom filter [46] containing the IDs of peers whose public keys are cached. The server communicates this Bloom filter to clients via the ServerHello message (specifically within the EncryptedExtensions message).

Upon receiving the Bloom filter, the client checks for its ID. If the client's ID is absent from the Bloom filter (cache miss indicated), it sends a hybrid-miss Looma signature that includes the WOTS$^+$ signature, its public key, the Merkle proof, a nonce, and a public key identifier, as depicted in Figure 5. If the client's ID is present in the Bloom filter (cache hit indicated), the client sends a streamlined hybrid-hit Looma signature containing only the WOTS$^+$ signature, a nonce, and the public key identifier.

Considering typical cloud communication patterns—where services interact frequently with a relatively small and fixed set of peers—a compact Bloom filter easily fits within `ServerHello` (see § VII-A).

In rare cases of false positives, where the Bloom filter indicates a cached key when it is not actually cached, the server detects the mismatch during verification, issues a `bad_offline_sig` alert, and gracefully falls back to a conventional re-handshake using standard Dilithium2 authentication. This approach ensures robustness, prevents connection loss, and retains performance in the typical, high-cache-hit scenarios.

D. Background Plane: Key Provisioning

To support seamless foreground operations, the background plane manages proactive distribution and retrieval of WOTS$^+$ key materials. Specifically, each service endpoint periodically uploads fresh batches of signed public keys to the *KeyDist* service, while also fetching updated public keys that have been uploaded by peer endpoints.

1) KeyDist Service: The *KeyDist* service is designed as a simple storage-based node, analogous to DNS servers. By minimizing trust assumptions, KeyDist does not rely on internal confidentiality or data-integrity guarantees beyond basic operational correctness and data availability. Each endpoint establishes a long-term secure channel with KeyDist, authenticated by standard (non-online/offline) signature schemes, since these connections rarely require re-authentication.

Cryptographic security instead stems from the individual endpoints. All public-key materials uploaded to KeyDist are digitally signed by their respective issuers using a strong signature scheme (Dilithium-2), independent of KeyDist. Microservice endpoints downloading these key materials subsequently verify their authenticity via embedded public keys from certificates, ensuring end-to-end integrity and authenticity even if KeyDist is compromised or misbehaving.

2) Key Generation: Each endpoint maintains logical groupings of peer services, called *verifier groups*, that share common sets of public keys. Initially, each endpoint creates a default group containing all peer services. Each verifier group has its own queue of WOTS$^+$ secret and public key pairs. When the queue size for a verifier group falls below a predetermined threshold S, the background plane proactively generates a fresh batch of WOTS$^+$ key pairs. These public keys are organized into a Merkle tree, whose root is signed using a Dilithium-2 private key (SK_{d2}). The endpoint then constructs a key record containing: (i) the list of public keys, (ii) the issuer's certificate, (iii) metadata such as verifier-group identification and key validity periods, and (iv) a Dilithium-2 signature computed over the hash of the entire public-key batch. The resulting record is uploaded to KeyDist in the format $\langle \mathsf{KeyUpdate}, \mathsf{keyrecord}, \mathsf{uploader\_id} \rangle$.

Upon receiving the upload, KeyDist performs several validation steps before storing the record: verifying the endpoint's certificate, reconstructing the Merkle tree structure, and validating the attached Dilithium-2 signature. Any failed validation results in the KeyDist service rejecting the update, ensuring that only authenticated and properly constructed public-key records are disseminated.

Adaptive Key Generation. Endpoints adaptively group their peer services based on interaction frequency. Peers that rarely communicate, termed *cold peers*, share a common pool of public keys. In contrast, peers involved in frequent interactions, called *hot peers*, each form individual verifier groups and receive dedicated key sets. This adaptive grouping prioritizes resource allocation towards frequently accessed keys, thereby optimizing memory usage, reducing unnecessary key-fetch operations, and maintaining efficient use of storage resources.

3) Key Fetching: Endpoints periodically request updated public keys from KeyDist by sending requests formatted as $\langle \mathsf{KeyFetch}, \mathsf{requester\_id} \rangle$. Since the endpoint's identity is authenticated when establishing the long-term TLS connection, KeyDist directly responds by returning the list of accessible keyrecord entries relevant to the requester's identity. Upon receipt, the endpoint independently verifies the authenticity of each received record and caches the validated public keys for subsequent foreground-plane operations.

VI. Security Analysis

We now analyze the security of the Looma authentication scheme, which combines a long-term Dilithium-2 key pair with per-handshake one-time WOTS$^+$ signatures. Specifically, we show that this construction achieves existential unforgeability under adaptive chosen-message attacks (EUF-CMA) [47]. We consider an adversary that may (i) trigger honest endpoints to perform the FastSign operation on chosen handshake transcripts and (ii) observe, replay, or inject arbitrary handshake messages. Our security goal is to preserve EUF-CMA with NIST level 1 post-quantum security, even if the KeyDist service becomes malicious or compromised.

Looma Security. To successfully forge a Looma signature, the adversary must produce a tuple $(HT^a, \sigma^a, PK^a, r^a)$ that passes verification: $\mathsf{FastVerify}(\mathsf{HT^a}, \sigma^a, \mathsf{PK^a}, \mathsf{r^a}) = 1$. Verification succeeds only if the following conditions are simultaneously satisfied:

1) *Dilithium-2 authentication:* PK^a is authenticated by a Merkle-path proof ending in a Dilithium-2 signature, which itself validates correctly under the legitimate endpoint's certificate.
2) *WOTS$^+$ authenticity:* σ^a is a valid WOTS$^+$ signature for the handshake transcript HT^a under the public key PK^a.
3) *Fresh-key enforcement:* PK^a is an unused one-time key that remains within its validity period, enforced via local indexing.

A successful forgery event implies the occurrence of at least one of the following security breaches:

E1. **Forgery of Dilithium-2 signature:** The adversary produces a malicious public key PK^a whose Merkle-tree root passes the Dilithium-2 check without a valid signature from the legitimate endpoint. This scenario constitutes a direct break of Dilithium-2's EUF-CMA security.

E2. **Forgery of WOTS$^+$ signature:** The adversary manages to generate a valid WOTS$^+$ signature σ^a under an authentic, legitimately distributed PK^a without knowing the corresponding secret key. This would violate the EUF-CMA security property of WOTS$^+$.

E3. **Malicious manipulation of KeyDist:** The adversary modifies stored keys on KeyDist. Nevertheless, since KeyDist stores only cryptographically signed key records, any attempt to inject malicious keys reduces directly to event E1 (forgery of the Dilithium-2 signature).

Because event E3 reduces directly to event E1, any successful forgery must directly violate either E1 or E2. Since the Dilithium-2 signature scheme is proven EUF-CMA-secure under standard lattice hardness assumptions [30], event E1 has negligible probability. Similarly, the WOTS$^+$ scheme has a formal EUF-CMA security proof relying on the second-preimage resistance, undetectability, and one-wayness of the underlying hash function family [35]. In our instance, we use the Haraka hash function [48], which satisfies all these cryptographic properties and has been widely studied in similar PQ signature contexts (e.g., SPHINCS$^+$ [36]). With parameters $n = 256$ bits and $w = 4$, our WOTS$^+$ instantiation achieves the same NIST level-1 (128-bit classical or approximately 64-bit quantum) security level as Dilithium-2.

Consequently, the adversary's forgery advantage is negligible, implying that the Looma scheme achieves the targeted EUF-CMA security.

KeyDist Security Analysis. Looma minimizes trust in the KeyDist service by treating it as an unauthenticated public repository for Dilithium-2–signed public keys. Endpoints verify all key records independently, ensuring that even a compromised or malicious KeyDist cannot forge or inject invalid keys. The worst-case impact is denial of service, not signature forgery. KeyDist's availability can be enhanced via standard cloud replication, making this design both secure and scalable for microservice-based infrastructures.

VII. Implementation

We present details regarding the implementation of Looma (§ VII-A), and its integration with TLS 1.3 (§ VII-B).

A. Looma Implementation

Our Looma implementation in C uses a custom version of WOTS$^+$, integrating three modern hash function families: SHA-256 [49] (from OpenSSL), BLAKE3 [50], and Haraka [48]. The performance of WOTS$^+$ varies significantly based on the chosen hash function and its parameters.

WOTS$^+$ Optimization. In WOTS$^+$, each public key is derived by iteratively hashing secret values $sk_i \in SK$ up to $w - 1$ times (§ III-C), with the Winternitz parameter w determining the length of these hash chains. For signing or verification, the number of hashes per element varies based on the message bits. By caching intermediate hash results during offline public key generation, we significantly accelerate the online signing phase (i.e., FastSign), converting it to fast memory-copy operations that takes less than 1μs. However, key generation and verification still involve extensive hashing operations on the critical path.

Choice of Hash Function. We benchmarked SHA-256, BLAKE3, and Haraka with WOTS$^+$ for signing and verifying 32-byte messages (see Table III). Haraka consistently delivered the best performance, outperforming SHA-256 and BLAKE3 by factors of 3–4. This performance advantage arises from Haraka's design, which specifically targets many small, fixed-size hashes. SHA-256 ranks second due to OpenSSL's highly optimized assembly routines and minimal overhead. In contrast, BLAKE3, despite its AVX2 acceleration, is optimized for longer messages where its parallelizable tree structure becomes advantageous, thus performing suboptimally with short inputs.

We further optimize Looma by using an eight-way SIMD-accelerated version of Haraka (Haraka8x), which leverages AES-NI intrinsics. By carefully structuring data for SIMD operations and minimizing branching, Haraka8x significantly reduces latency in both key generation and verification.

TABLE III: Signature size and per-operation latency of WOTS+ signature across three modern hash families.

w	ℓ	Sig/PK/SK size (B)	Key-gen (µs)				Sign (µs)	Verify (µs)			
			SHA256	BLAKE3	Haraka	Haraka8x		SHA256	BLAKE3	Haraka	Haraka8x
2	265	8,480	35.57	47.23	11.90	7.18	0.62	20.45	23.12	6.43	4.60
4	133	4,256	51.17	68.60	15.26	10.10	0.38	26.82	36.33	8.64	6.43
8	90	2,880	79.42	106.49	22.94	15.42	0.29	72.86	95.33	19.93	13.98
16	67	2,144	128.01	169.33	35.73	24.61	0.25	58.89	83.77	17.16	13.50
32	55	1,760	213.03	290.00	60.45	43.09	0.24	198.58	271.58	52.60	38.63
64	45	1,440	353.73	475.80	99.21	71.07	0.21	308.46	434.51	97.51	70.12

Choice of Winternitz Parameter (w). The parameter w controls the trade-off between signature size and computational effort in WOTS+. Larger values of w compress signatures but increase computational latency due to longer hash chains. Conversely, smaller values yield faster computation but larger signatures. As shown in Table III, increasing from $w = 2$ to $w = 4$ results in only a modest latency penalty (approximately 1.8μs) but significantly reduces signature size by about 50%. However, increasing w further to 8 triples the verification latency relative to $w = 4$, rendering it impractical for latency-sensitive scenarios. Thus, our optimal configuration adopts Haraka8x with $w = 4$, balancing compactness and computational efficiency.

Bloom Filter Configuration. Based on the data points in § IV-A, we size our Bloom filter for 10 peers—a target that matches the 99% burst in microservices, and still leaves headroom for infrequent outliers. We use the 32-bytes bitmap length and 13 non-crypto hash probes for our Bloom filter, expecting $p \approx 0.01\%$ false positive rate. The CPU cost is negligible, only tens of nanoseconds.

B. Integration with TLS

To evaluate candidate PQ signature algorithms, we rely on the `OQS provider`[51] available in `OpenSSL 3.2.0`. To comprehensively evaluate our design, we incorporate classical signatures, PQ candidates, and our own Looma into Picotls [52], a lightweight and modular TLS 1.3 implementation leveraging `OpenSSL` as its cryptographic backend. Picotls is selected over direct modification of `OpenSSL` due to its well-defined support for custom authentication callbacks and efficient reuse of `OpenSSL`'s optimized cryptographic primitives. This modular approach enables clean benchmarking and rapid iteration of handshake variants without impacting legacy protocol components or the broader OpenSSL library.

Our implementation is transparent at the application level, allowing any existing Picotls-based application to adopt our enhanced PQ authentication without source modifications. Picotls is already widely adopted in HTTP/3 deployments [53], [54], offering a realistic testbed for practical evaluation and straightforward extension of our PQ security enhancements to modern transport protocols such as QUIC [55].

VIII. Evaluation

We evaluate Looma with regular TLS and mutual TLS handshakes in comparison to other candidate signature schemes.

TABLE IV: Hardware and OS details.

Server-A2	AMD EPYC 9555P CPU @ 384 GB RAM
Server-I2	2×Intel Xeon Gold 5418N CPUs @ 128 GB RAM
NIC/Switch	Mellanox ConnectX-7 / NVIDIA SN3700 100Gbps
Software	Ubuntu 24.04.1 LTS with Linux 6.8.0-49-generic

Testbed. We use two pairs of identical machines—denoted A2 and I2—with their configurations detailed in Table IV. Although we discuss key results with both setups in this section, due to space limit, plots for I2 experiment appear in Appendix A. The KeyDist service runs on a separate third machine. Network bandwidth is never the limiting factor. We use the default MTU of 1,500 B (common in public clouds such as AWS and Azure)[2]. We use `clock_gettime(CLOCK_MONOTONIC)` to timestamp relevant events accurately.

Baselines. We compare Looma against NIST's PQ schemes—Dilithium-2, Falcon-512, and SPHINCS+—and against classic signatures: RSA-2048 and ECDSA over `secp256r1`. We also tested Ed25519, but its performance closely matches ECDSA and is omitted for clarity. We target 128-bit classical security and NIST Level 1 post-quantum security. Note that RSA-3072 offers 128-bit classical security, but we show RSA-2048 due to its wide deployment and faster signature operations.[3] RSA and ECDSA/Ed25519 would offer 0 bits of security in a post-quantum setting.

Looma Configuration. Looma employs WOTS+ with Haraka8x and Winternitz parameter $w = 4$. A dedicated CPU core runs the background plane to sustain key provisioning throughput. We use a 32-byte Bloom filter with 13 probes and a Merkle tree of size 1024.

TLS Setup. All experiments use full TLS 1.3 handshakes in 1-RTT mode. We exclude PSK-resumption handshakes, as they bypass authentication. We evaluate both server-side authenticated TLS (*sTLS*) and mutual TLS (*mTLS*), with X.509 certificates issued by a CA. The key-exchange method is uniform across schemes: X25519 ECDH on `Curve25519`. Classic certificates are signed with RSA by the CA; PQ certificates use Dilithium-2. We do not model certificate revocation, assuming short-lived certs to avoid CRL or OCSP overhead.

[2]With a larger MTU (e.g., 9 KB), Looma 's advantages increase slightly, since ServerHello messages fit into one packet, reducing host-stack overheads.

[3]On our A2 testbed, RSA-3072 sTLS/mTLS handshakes average 860.2/1966.7 μs, with sign/verify operations costing 570.5/29.3 μs.

TABLE V: TLS handshake latency summary (μs).

Signature Algorithm	sTLS				mTLS			
	Mean	St. Dev.	P50	P99	Mean	St. Dev.	P50	P99
RSA-2048	842.4	206.3	900	1,427	1,095.4	179.2	1,112	1,656
ECDSA-256	422.0	38.9	416	513	462.6	38.6	449	555
ED25519	417.6	47.4	411	511	429.3	57.0	432	534
Dilithium-2	499.9	106.2	485	837	560.5	112.4	548	902
Falcon-512	778.9	117.6	801	902	969.9	56.6	963	1,066
SPHINCS$^+$ SHA256-128f-simple	13,358.5	3,611.7	12,725	18,894	35,289.7	1,264.3	35,471	36,099
Looma	**354.9**	**42.3**	**348**	**447**	**363.6**	**46.2**	**363**	**464**

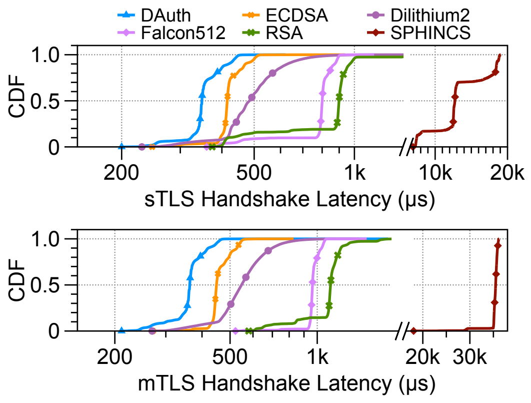


Fig. 6: sTLS (top) and mTLS (bottom) handshake-latency CDFs. (Summary in Table V).

A. Basic Handshake Latency

We measure the *end-to-end TLS handshake latency* of Looma and several baselines by timing from the `ClientHello` to the completion of the handshake, following the methodology of [56]. This metric reflects the latency a client incurs before establishing a secure tunnel, excluding the TCP handshake. For each signature scheme, the client and server perform 10,000 sequential connections. To mitigate cache-warm effects, we precede the main loop with a client-side warm-up phase and randomize the order in which algorithms are tested.

Figure 6 plots the cumulative distribution function of handshake latencies, and Table V reports mean and key percentiles for the A2 setup. In the sTLS case (top), Looma outperforms the fastest classic baseline, ECDSA, by 32 % at the 50th percentile (P50) and 11 % at the 99th percentile (P99). Against the fastest PQ baseline, Dilithium-2, Looma reduces latency by 37 % at P50 and 42 % at P99. In the mTLS scenario (bottom), Looma achieves 16 % and 14 % improvements over ECDSA at P50 and P99, respectively, and 20 % and 33 % improvements over Dilithium-2. Falcon-512 and RSA incur higher latencies in both modes, while SPHINCS$^+$ is markedly slower and thus omitted from throughput comparisons.

Results in the I2 setup exhibit a similar trend, with all schemes experiencing approximately 2.5× higher handshake latencies (see Figure 10 and Table X in Appendix A).

TABLE VI: Sign and verification latency (in μs).

Algorithm	I2		A2	
	Sign	Verify	Sign	Verify
RSA-2048	261.1	19.3	200.7	15.4
ECDSA-256	21.1	62.6	14.9	40.8
Ed25519	27.6	85.2	18.7	54.4
Dilithium-2	61.4	24.4	51.4	21.3
Falcon-512	181.1	34.3	149.2	28.3
SPHINCS$^+$-SHA2-128f	6866.2	524.9	5611.6	426.3
Looma	**0.39**	**6.83**	**0.32**	**6.19**

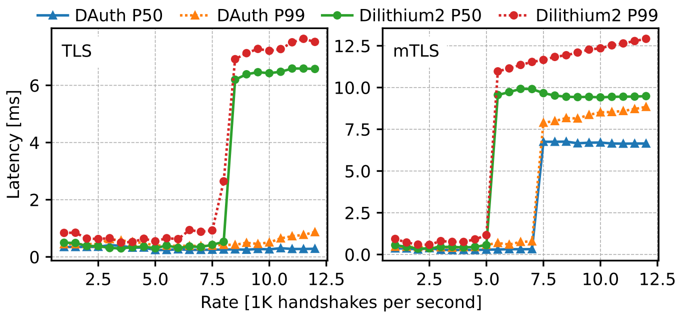


Fig. 7: Handshake latency over increasing request rates for sTLS (left) and mTLS (right).

B. Authentication Latency

To pinpoint the source of Looma 's low handshake latency, we measure the per-operation cost of each signature scheme. We execute 100,000 signing and verification operations, reporting the averages for both A2 and I2 setups in Table VI. Across all schemes, Intel-based nodes exhibit per-operation times approximately 1.2–1.5× those on AMD machines, highlighting the influence of microarchitectural differences on cryptographic performance. Both results confirm that Looma substantially reduces both signing and verification overhead. This improvement allows Looma to outperform classic ECDSA—whose verification is notably more expensive than signing—and the PQ baseline Dilithium-2. These timings align closely with our implementation strategy (see § VII), validating the efficiency gains of the online/offline paradigm.

C. Concurrent Handshake Latency

We next evaluate end-to-end handshake latency under varying request rates and, as a result, server load. High request rates create queues at the server software stack, increasing the latency. In this experiment, the server uses a single thread while the client spawns as many threads as needed to issue and process the handshake requests at the target rate.

Figure 7 plots P50 and P99 latency of Looma and Dilithium-2, the fastest PQ baseline. As the request rate approaches the maximum capacity, the P50 and P99 latencies begin to increase, and their spike indicates that the rate has reached the limit. Looma maintains lower latency than Dilithium-2 and sustains near-basic handshake latency (see § VIII-A) up to higher request rates for both sTLS and mTLS. Dilithium-2 fails to meet the target rate beyond 8 kRPS in sTLS case and 5.5 kRPS in mTLS case, whereas Looma continues to serve requests, showing small P99 latency increase in sTLS or peaking at 7.5 kRPS in mTLS.

D. Peak Throughput

Having demonstrated Looma 's latency benefits, we now evaluate its peak handshake throughput under two cloud-relevant scenarios: many clients stressing a single server (many-to-one) and a single client contacting many servers (one-to-many).

1) Many-to-One Scenario: Here, we describe the many-to-one setup. In this experiment, the server runs four threads while the client scales from four to 32 threads to generate load. Each client thread issues 20 concurrent TLS/TCP handshakes, closing each connection immediately after completion. Figure 8 (top: sTLS; bottom: mTLS) plots throughput and latency as client threads increase in A2 setup, and results in I2 setup can be found in Figure 12 in Appendix. In both sTLS and mTLS, as the server has already started being saturated at 8 threads, latency starts increasing with that number of threads due to request backlogs.

a) sTLS: In A2, Looma outperforms the best PQ baseline, Dilithium-2, by 4 % to 13 % in throughput and reduces P50 and P99 latency by 24 % to 38 % and 3 % to 11 %, respectively. In I2, throughput gains reach 6 % to 37 %, with latency down by 13 % to 29 % at P50 and −16 % to 35 % at P99. The negative P99 latency improvement with 4 threads arises because Looma serves 36 % more requests with a larger request backlog. Nevertheless, we observe that Looma simultaneously improves throughput and latency in most cases.

b) mTLS: Under mTLS, Looma achieves 8 % to 22 % higher throughput in A2 and 29 % to 36 % in I2 versus Dilithium-2. P50 latency drops by 29 % to 34 % (A2) and 25 % to 31 % (I2); P99 latency improves by −21 % to 18 % and 21 % to 32 %, respectively, with the lone negative case again due to higher throughput at low thread counts. ECDSA shows a similar pattern, with negative P99 improvements only against slower schemes.

Overall, these results confirm that Looma delivers superior handshake throughput without sacrificing latency.

2) One-to-Many Scenario: Next, we evaluate the scenario where a single client concurrently authenticates with many servers. To emulate this, the client uses one thread while each of 32 server threads accepts connections; the client then increases its number of simultaneous TCP/TLS handshakes, creating a backlog at the client side.

a) sTLS: Figure 9 (top) shows sTLS throughput and latency for the A2 setup (I2 trends are similar). Looma outperforms the strongest PQ baseline, Dilithium-2, by 6 % to 23 % (A2) and 7 % to 21 % (I2) in throughput. P50 latency improves by 5 % to 23 % (A2) and 13 % to 22 % (I2), and P99 by 12 % to 33 % (A2) and 21 % to 30 % (I2). We also observe that ECDSA and Falcon-512 perform worse under heavy sTLS load; because the client only verifies signatures in sTLS, verification dominates latency, and ECDSA/Falcon-512 have the slowest verify times (see Table VI).

b) mTLS: Figure 9 (bottom) presents mTLS results. Here, Looma's throughput gains over Dilithium-2 grow to 40 % to 52 % (A2) and 27 % to 42 % (I2). P50 latency drops by 35 % to 44 % (A2) and 24 % to 36 % (I2), and P99 by 34 % to 42 % (A2) and 35 % to 41 % (I2).

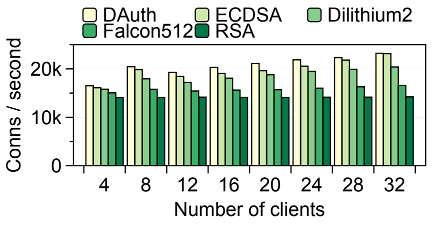
The larger mTLS improvements stem from increased client-side signing: mTLS requires both signing and verification, and post-quantum signing is significantly more expensive than verification (see Table VI). Looma 's online/offline optimization therefore yields greater relative gains when signing dominates. In contrast, the many-to-one scenario was server-bound, so Looma 's verification efficiency drove similar improvements in both sTLS and mTLS.

E. Key-Provisioning Overhead

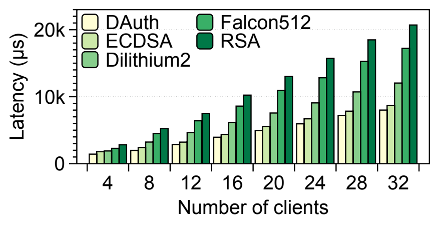
1) KeyDist: KeyDist organizes pre-signed keys into batches of 1,024 keys stored as 4 MB files. We implement the KeyDist server using Nginx [57] to serve these files and evaluate its provisioning capacity by issuing requests from another machine over multiple concurrent connections, emulating many endpoints. The KeyDist server runs on a machine equipped with an Intel Xeon Silver 4314 CPU and a Samsung PM9A3 NVMe SSD, while clients run on one of the A2 machines. Under a workload of 128 concurrent connections, we confirmed that the server sustains 2.69 K file requests per second, corresponding to 21.5 K keys per second per client. Since the maximum key-consumption rate at a single endpoint is 2.75 K keys per second (derived from the average Looma mTLS handshake latency in Table V), these results indicate that a single KeyDist instance can comfortably support hundreds of endpoints even under aggressive key usage. To sustain this rate for 20 minutes, a KeyDist server requires approximately 1.24 TiB of key material. This space can be reclaimed after the corresponding keys are consumed or expire. For reference, the NVMe SSD we use costs $241.82 per TiB.

2) Endpoints: We evaluate the computational and storage overhead at each endpoint as follows.

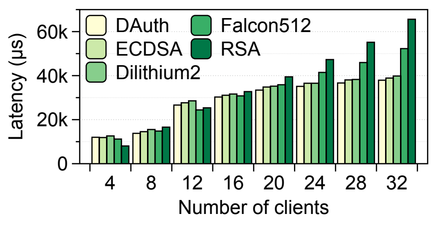
a) KeyGen: We found that each endpoint can generate keys extremely efficiently—about 41 000 key pairs per second on a single dedicated CPU core with our Looma configuration. A full KeyGen cycle (roughly 24.4 ms) for a batch of 1024 key pairs includes (i) key-pair generation, (ii) hashing 1023 internal nodes, (iii) one Dilithium-2 root signature, and (iv) computing 1024 inclusion proofs. That works out to just



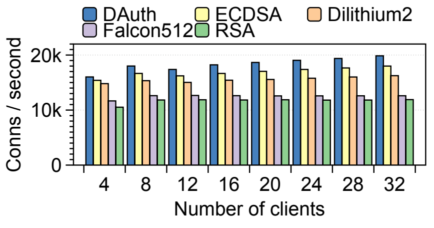
(a) Throughput (sTLS)



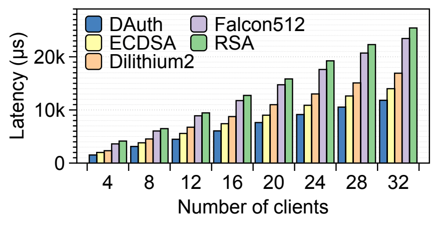
(b) P_{50} latency (sTLS)



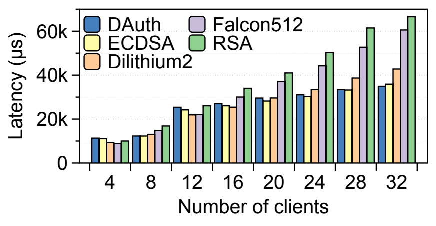
(c) P_{99} latency (sTLS)



(d) Throughput (mTLS)

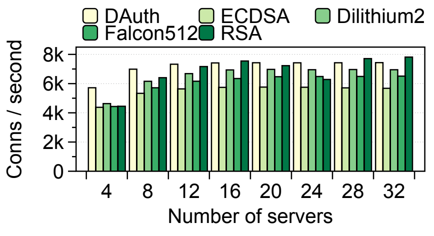


(e) P_{50} latency (mTLS)

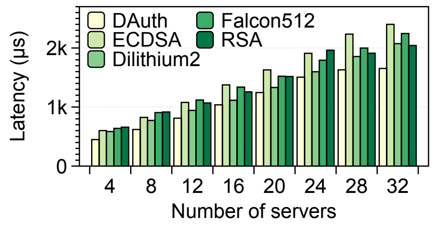


(f) P_{99} latency (mTLS)

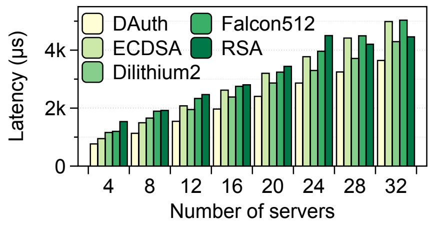
Fig. 8: Handshake throughput and latency as the number of concurrent clients increases. Looma maintains low latency while delivering the highest throughput.



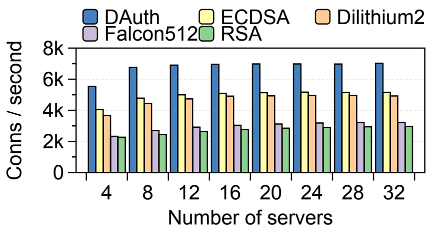
(a) Throughput (sTLS)



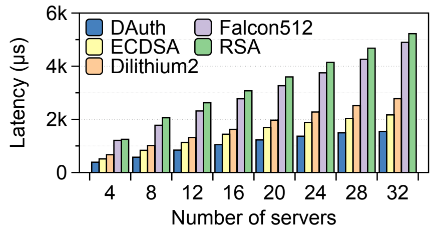
(b) P_{50} latency (sTLS)



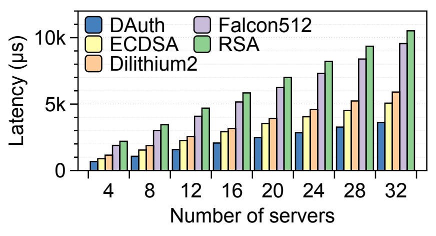
(c) P_{99} latency (sTLS)



(d) Throughput (mTLS)



(e) P_{50} latency (mTLS)



(f) P_{99} latency (mTLS)

Fig. 9: Handshake throughput and latency as the number of concurrent servers increases. Looma sustains low latency even while scaling throughput.

23.8 μs per key pair on average, which aligns perfectly with the results in Table III. There, WOTS^{+} key generation is the dominant contributor to latency, while the costs of tree construction and root signing are spread across many keys. The offline artifacts occupy ($\approx$ 17 MiB) in total.

b) KeyFetch: Refreshing key records from KeyDist requires one Dilithium-2 verification plus Merkle reconstruction over 1024 leaves, for a total of 0.33 ms. After validation, only the 1024 public keys ($\approx$ 4.16 MiB) are retained.

c) Impact: Both tasks run off-path. The only visible delay occurs during initial instance bootstrap ($\approx$ 24 ms), which is negligible compared to typical cloud-service startup times. The memory footprint (17 MiB for FastSign, 4 MiB for FastVerify) represents under 2% of a 1 GiB RAM allocation and under 1% of a 2 GiB container—acceptable overhead in modern cloud deployments. Transmission costs on multi-gigabit links are insignificant.

F. Cache Misses

We evaluate the performance of two fallback strategies under cache-miss conditions. Recall that cache misses are specific to mTLS, because they arise only when the server fails to refresh a client's public key in time and thus has to process a dual signature in hybrid or dual-sig mode (see § V-C). Table VII reports the handshake latency for cache-hit and -miss cases in A2.

TABLE VII: Handshake latency with cache misses (in μs). Hybrid/Hit case is identical to the mTLS case in Table V

		mTLS				Detail	
Fallback	Hit/Miss	Mean	St. Dev.	P50	P99	Sign	Verify
Hybrid	Hit	363.6	46.2	363	464	0.32	6.19
	Miss	367.7	48.3	365	468	0.32	29.72
Dual-sig	Hit	365.7	48.0	365	462	0.32	6.12
	Miss	367.3	45.7	366	460	0.32	29.51

To understand this, we microbenchmark the signing and verification costs and observe that verifying a dual signature adds only about 23 µs compared to verifying a WOTS$^+$ signature in the cache-hit case. This gap is consistent with the cost of verifying a Dilithium-2 signature (see Table VI). Note that both cache-hit and cache-miss cases benefit from the fast signing time enabled by WOTS$^+$ (about 0.32 µs), in sharp contrast to Dilithium-2, which requires 50 µs to 60 µs per signing operation. Consequently, Looma maintains a substantial latency advantage over Dilithium-2 and other baselines even when cache misses occur.

IX. Discussion

Applicability to Hybrid PQTLS and KEMTLS. Looma accelerates the signature-based authentication step in TLS 1.3 without altering the key exchange mechanism, making it compatible with classical ECDHE, hybrid ECDHE+KEM, and pure KEM-based exchanges. In our evaluation, we fixing the key exchange to ECDHE does not imply that Looma requires classical key exchange. Hybrid PQTLS [58], [59], [60] is a transitional approach that combines classical ECDHE with post-quantum KEMs in TLS 1.3 key exchange, while retaining flexibility in the authentication phase (e.g., using classical or post-quantum signatures). Looma can substitute traditional signature algorithms such as RSA/ECDSA or post-quantum signatures (e.g., Dilithium, Falcon) with its online/offline variant under our design. With appropriate configuration, Looma integrates seamlessly with hybrid PQTLS, reducing authentication overhead in both classical and post-quantum settings. However, Looma is not applicable to KEMTLS [27], which replaces signature-based authentication with KEM-based endpoint authentication.

Applicability to Other Encryption Protocols. While we have evaluated Looma in the context of regular TLS and mTLS over TCP, its benefits extend to other TLS-type handshake mechanism. For example, datapath encryption frameworks like Google's PSP [61] and kernel-TLS (kTLS) both perform the TLS 1.3 handshake in user space; replacing their signing paths with Looma's online/offline paradigm would directly reduce their connection-setup latency.

The advantage of Looma is even more pronounced in protocols that already shave off a round trip. QUIC—increasingly adopted in serverless platforms for faster function invocation [62]—eliminates one RTT compared to TLS/TCP, so per-handshake signing overhead becomes dominant. By integrating Looma into QUIC (e.g., via Picotls), one could recover much of that cost, yielding faster secure channel establishment.

Offload Support. By decoupling expensive signing computations, Looma naturally enables offload to accelerator devices. SmartNICs [22] or remote TLS proxies with special purpose CPU cores [23] can pre-compute and verify WOTS key material and inclusion proofs outside the handshake process on the host CPUs.

X. Related Work

We have already discussed TLS handshake acceleration in § II-B and signature schemes in § III. We discuss the rest here.

a) TLS datapath acceleration: TLS acceleration for user data transfer (i.e., after the handshake) has been widely explored, and those techniques are complementary to Looma. Facebook introduced offloading TLS to the kernel in datacenters [63], while keeping the handshake in user-space and letting the application to register the negotiated keys to the kernel for subsequent cryptographic operations, which is now called kTLS [64]. Cisco/Cilium uses it for their network observability service [65], [66]. NVIDIA NICs support offloading kTLS processing in the NIC [67].

b) Datacenter-friendly signature: The recently proposed DSig [68] introduces a microsecond-scale hybrid signature implementation and apply it in datacenter distributed systems that manage frequent transactions and prioritize auditability, such as Byzantine fault-tolerant (BFT) broadcast systems. It achieves low latency and high throughput by letting a singer pre-send a list of hash-based public keys (signed by a traditional digital signature scheme EdDSA) to the verifier, which allows a verifier pre-verify the public key list before it receives a signed message. It inspires our design to deploy such online-offline paradigm into datacenter TLS.

c) Related work beyond TLS 1.3: Several systems pursue conceptually similar ideas—shifting computational cost or leveraging offline work—but target different protocols or goals than TLS 1.3 handshake authentication. Reverse SSL [69] protects TLS 1.2 servers against DoS attacks by using an online/offline RSA-based signing approach to reduce server-side workload, while introducing client puzzles to throttle adversaries—at the expense of increased client-side handshake latency. Waters et al. [70] propose outsourcing and precomputing puzzle effort to improve DoS resilience in general networked systems, not TLS specifically; their techniques apply across layers (e.g., IP/TCP or application protocols) to redistribute computational burden. Delegated Credentials [71] enable a certificate holder to delegate TLS authentication by signing short-lived keys, thereby limiting key exposure. This sign-and-distribute pattern also appears in our key provisioning design, which similarly issues time-bounded public keys to balance security and performance.

XI. Conclusion

To cope with the computational overheads posed by modern security requirements in datacenters—mutual authentication and post-quantum cryptography, this paper explored a new approach to fast TLS handshake. Based on the observation that the post-quantum cryptography exhibits costly sign and cheap verify performance characteristics, we designed, implemented and evaluated a new digital signature architecture, Looma. We will release the all the code used in our experiment upon the acceptance of the paper.

XII. Ethics Considerations

We have considered the risks and benefits of this research and, to the best of our knowledge, it raises no ethical concerns. All experiments were conducted on isolated testbeds using open-source software; no human subjects, user data, or sensitive information were involved.

Acknowledgement

This work was in part supported by EPSRC grant EP/V053418/1, Royal Society Research Grant, and gift from Google and NetApp.

References

[1] D. of Physics at Oxford, "New world record for qubit operation accuracy," 9 June 2025. [Online]. Available: https://www.physics.ox.ac.uk/news/new-world-record-qubit-operation-accuracy

[2] D. Castelvecchi, "'A truly remarkable breakthrough': Google's new quantum chip achieves accuracy milestone." Dec. 2024. [Online]. Available: https://www.nature.com/articles/d41586-024-04028-3

[3] H. Neven, "Meet willow, our state-of-the-art quantum chip," Dec. 2024. [Online]. Available: https://blog.google/technology/research/google-willow-quantum-chip/

[4] K. John, "Ibm starling: 20,000x faster than today's quantum computers," Jun. 2025. [Online]. Available: https://www.forbes.com/sites/johnkoetsier/2025/06/10/ibm-starling-20000x-faster-than-todays-quantum-computers/

[5] C. Gidney, "How to factor 2048 bit rsa integers with less than a million noisy qubits," 2025. [Online]. Available: https://arxiv.org/abs/2505.15917

[6] M. Campagna, "Hybrid-key exchanges as an interim-to-permanent solution to cryptographic agility," 2019.

[7] K. Kwiatkowski, "Towards Post-Quantum Cryptography in TLS," https://blog.cloudflare.com/towards-post-quantum-cryptography-in-tls/, June 2019.

[8] K. Kwiatkowski and L. Valenta, "The TLS Post-Quantum Experiment," https://blog.cloudflare.com/the-tls-post-quantum-experiment/, Oct. 2019.

[9] A. Langley, "CECPQ1 results," https://www.imperialviolet.org/2016/11/28/cecpq1.html, Nov. 2016.

[10] ——, "CECPQ2," https://www.imperialviolet.org/2018/12/12/cecpq2.html, Dec. 2018.

[11] ——, "Post-quantum confidentiality for TLS," https://www.imperialviolet.org/2018/04/11/pqconftls.html, April. 2018.

[12] Twitter, "Rebuilding twitter's public api," https://blog.x.com/engineering/en_us/topics/infrastructure/2020/rebuild_twitter_public_api_2020, 2020.

[13] Uber, "Rewriting uber engineering: The opportunities microservices provide," https://www.uber.com/en-GB/blog/building-tincup-microservice-implementation/, 2016.

[14] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 283–298.

[15] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious {Multi-Party} machine learning on trusted processors," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 619–636.

[16] D. Dehigama, S. Jesalpura, A. Katsarakis, M. Kogias, R. Kumar, and B. Grot, "Composing microservices and serverless for load resilience," in *The 2nd Workshop on SErverless Systems, Applications and MEthodologies*, 2024.

[17] H. Saokar, S. Demetriou, N. Magerko, M. Kontorovich, J. Kirstein, M. Leibold, D. Skarlatos, H. Khandelwal, and C. Tang, "ServiceRouter: Hyperscale and minimal cost service mesh at meta," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 969–985. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/saokar

[18] V. Addanki, O. Michel, and S. Schmid, "PowerTCP: Pushing the performance limits of datacenter networks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 51–70. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/addanki

[19] Y. Liu, W. Li, Y. Li, L. Suo, X. Gao, X. Xie, S. Chen, Z. Fan, W. Qu, and G. Liu, "Fork: A dual congestion control loop for small and large flows in datacenters," in *Proceedings of the Twentieth European Conference on Computer Systems*, ser. EuroSys '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 446–459. [Online]. Available: https://doi.org/10.1145/3689031.3696101

[20] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 63–74. [Online]. Available: https://doi.org/10.1145/1851182.1851192

[21] K. Jang, S. Han, S. Han, S. Moon, and K. Park, "{SSLShader}: Cheap {SSL} acceleration with commodity processors," in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[22] D. Kim, S. Lee, and K. Park, "A case for smartnic-accelerated private communication," in *Proceedings of the 4th Asia-Pacific Workshop on Networking*, 2020, pp. 30–35.

[23] E. Song, Y. Song, C. Lu, T. Pan, S. Zhang, J. Lu, J. Zhao, X. Wang, X. Wu, M. Gao *et al.*, "Canal mesh: A cloud-scale sidecar-free multi-tenant service mesh architecture," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024, pp. 860–875.

[24] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8446

[25] P. Eronen, H. Tschofenig, H. Zhou, and J. A. Salowey, "Transport Layer Security (TLS) Session Resumption without Server-Side State," RFC 5077, Jan. 2008. [Online]. Available: https://www.rfc-editor.org/info/rfc5077

[26] S. Lim, H. Lee, H. Kim, H. Lee, and T. Kwon, "Ztls: A dns-based approach to zero round trip delay in tls handshake," in *Proceedings of the ACM Web Conference 2023*, 2023, pp. 2360–2370.

[27] P. Schwabe, D. Stebila, and T. Wiggers, "Post-quantum tls without handshake signatures," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1461–1480.

[28] J. Zhang, J. Huang, L. Zhao, D. Chen, and Ç. K. Koç, "{ENG25519}: Faster {TLS} 1.3 handshake using optimized x25519 and ed25519," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 6381–6398.

[29] D. J. Bernstein, B. B. Brumley, M.-S. Chen, and N. Tuveri, "{OpenSSLNTRU}: Faster post-quantum {TLS} key exchange," in *31st USENIX security symposium (USENIX Security 22)*, 2022, pp. 845–862.

[30] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-dilithium: A lattice-based digital signature scheme." *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 1, pp. 238–268, 2018.

[31] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, Z. Zhang *et al.*, "Falcon: Fast-fourier lattice-based compact signatures over ntru," *Submission to the NIST's post-quantum cryptography standardization process*, vol. 36, no. 5, pp. 1–75, 2018.

[32] D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, M. Schneider, P. Schwabe, and Z. Wilcox-O'Hearn, "Sphincs: practical stateless hash-based signatures," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2015, pp. 368–397.

[33] S. Even, O. Goldreich, and S. Micali, "On-line/off-line digital signatures," in *Conference on the Theory and Application of Cryptology*. Springer, 1989, pp. 263–275.

[34] ——, "On-line/off-line digital signatures," *Journal of Cryptology*, vol. 9, no. 1, pp. 35–67, 1996.

[35] A. Hülsing, "W-ots+–shorter signatures for hash-based signature schemes," in *Progress in Cryptology–AFRICACRYPT 2013: 6th International Conference on Cryptology in Africa, Cairo, Egypt, June 22-24, 2013. Proceedings 6*. Springer, 2013, pp. 173–188.

[36] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, "The sphincs+ signature framework," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 2129–2146.

[37] A. Huelsing, D. Butin, S.-L. Gazdag, J. Rijneveld, and A. Mohaisen, "XMSS: eXtended Merkle Signature Scheme," RFC 8391, May 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8391

[38] F. Du, J. Shi, Q. Chen, P. Pang, L. Li, and M. Guo, "Generating microservice graphs with production characteristics for efficient resource scaling," 2025.

[39] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4152–4166, 2021.

[40] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on information theory*, vol. 29, no. 2, pp. 198–208, 2003.

[41] S. Frankel and S. Krishnan, "IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap," RFC 6071, Feb. 2011. [Online]. Available: https://www.rfc-editor.org/info/rfc6071

[42] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8446

[43] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9000

[44] A. Shamir and Y. Tauman, "Improved online/offline signature schemes," in *Annual International Cryptology Conference*. Springer, 2001, pp. 355–367.

[45] M. R. Albrecht, N. Gama, J. Howe, and A. K. Narayanan, "Post-quantum online/offline signatures," Cryptology ePrint Archive, Paper 2025/117, 2025. [Online]. Available: https://eprint.iacr.org/2025/117

[46] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[47] S. Goldwasser, S. Micali, and R. L. Rivest, "A digital signature scheme secure against adaptive chosen-message attacks," *SIAM Journal on computing*, vol. 17, no. 2, pp. 281–308, 1988.

[48] S. Kölbl, M. M. Lauridsen, F. Mendel, and C. Rechberger, "Haraka v2–efficient short-input hashing for post-quantum applications," *IACR Transactions on Symmetric Cryptology*, pp. 1–29, 2016.

[49] W. Penard and T. Van Werkhoven, "On the secure hash algorithm family," *Cryptography in context*, pp. 1–18, 2008.

[50] J.-P. Aumasson, S. Neves, J. O'Connor, and Z. Wilcox, "The BLAKE3 Hashing Framework," Internet Engineering Task Force, Internet-Draft draft-aumasson-blake3-00, Jul. 2024, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-aumasson-blake3/00/

[51] O. Q. Safe, "Oqs provider," https://github.com/open-quantum-safe/oqs-provider.

[52] G. repo, "H2O-picotls project," https://github.com/h2o/picotls.

[53] H2O, "quickly," https://github.com/h2o/quicly.

[54] Private-octopus, "Picoquic," https://github.com/private-octopus/picoquic.

[55] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9000

[56] D. Sikeridis, P. Kampanakis, and M. Devetsikiotis, "Post-quantum authentication in tls 1.3: A performance study," *Cryptology ePrint Archive*, 2020.

[57] I. Sysoev and I. NGINX, "nginx web server," https://nginx.org/, accessed: 27 November 2025.

[58] T. Reddy.K and H. Tschofenig, "Post-Quantum Cryptography Recommendations for TLS-based Applications," Internet Engineering Task Force, Internet-Draft draft-ietf-uta-pqc-app-00, Sep. 2025, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-uta-pqc-app/00/

[59] D. Stebila, S. Fluhrer, and S. Gueron, "Hybrid key exchange in TLS 1.3," Internet Engineering Task Force, Internet-Draft draft-ietf-tls-hybrid-design-16, Sep. 2025, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-tls-hybrid-design/16/

[60] Amazon Web Services, "Using hybrid post-quantum TLS with AWS KMS," https://docs.aws.amazon.com/kms/latest/developerguide/pqtls.html, Nov. 2019.

[61] N. Dukkipati, N. Bansod, C. Zhao, Y. Li, J. Bhat, S. Saleem, and A. S. Jain, "Falcon: A reliable and low latency hardware transport," The Technical Conference on Linux Networking (Netdev 0x18), https://netdevconf.info/0x18/sessions/talk/introduction-to-falcon-reliable-transport.html, 2024.

[62] K. Hou, S. Lin, Y. Chen, and V. Yegneswaran, "Qfaas: accelerating and securing serverless cloud networks with quic," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 240–256.

[63] T. Herbert, "Data center networking stack," The Technical Conference on Linux Networking (Netdev 1.2), https://legacy.netdevconf.info/1.2/session.html?tom-herbert/, 2016.

[64] "Kernel tls offload," https://www.kernel.org/doc/html/latest/networking/tls-offload.html.

[65] J. Fastabend, "Seamless transparent encryption with bpf and cilium," Linux Plumbers Conference 2019.

[66] D. Borkmann and J. Fastabend, "Combining ktls and bpf for introspection and policy enforcement," Linux Plumbers Conference 2018.

[67] B. Pismenny, H. Eran, A. Yehezkel, L. Liss, A. Morrison, and D. Tsafrir, "Autonomous nic offloads," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 18–35.

[68] M. K. Aguilera, C. Burgelin, R. Guerraoui, A. Murat, A. Xygkis, and I. Zablotchi, "DSig: Breaking the barrier of signatures in data centers," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, jul 2024, pp. 667–685. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/aguilera

[69] K. Bicakci, B. Crispo, and A. S. Tanenbaum, "Reverse ssl: Improved server performance and dos resistance for ssl handshakes," IACR Cryptology ePrint Archive, Tech. Rep. 2006/212, 2006. [Online]. Available: https://eprint.iacr.org/2006/212

[70] B. Waters, A. Juels, J. A. Halderman, and E. W. Felten, "New client puzzle outsourcing techniques for DoS resistance," in *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*. Washington, DC, USA: Association for Computing Machinery, 2004, pp. 246–256.

[71] R. Barnes, S. Iyengar, N. Sullivan, and E. Rescorla, "Delegated Credentials for TLS and DTLS," RFC 9345, Jul. 2023. [Online]. Available: https://www.rfc-editor.org/info/rfc9345

Appendix

This section provides plots and tables whose A2 setup results have been presented in § VII or § VIII. Table IX shows the storage space and latency trade-off of WOTS$^+$ signature scheme. Figure 10 plots the handshake latency whose results in the A2 setup has been presented in Figure 6. Table IX shows WOTS$^+$ storage and computational cost analysis whose results in the A2 setup have been presented with Table III in § VII. Table X shows the basic handshake latency whose results in the A2 setup have been shown in Table V. Figure 12 and Figure 13 correspond to Figure 8 and Figure 9, respectively. Table XI shows the end-to-end latency with cache misses in I2.

TABLE VIII: Online/offline signature frameworks

Framework	**Idea**	**Pros / cons**
Even–Goldreich–Micali (EGM) [33]	Pre-sign a one-time signature (OTS) on a random value offline; in the online phase sign that value with a long-term scheme.	Very simple; relies only on hash-based OTS, but the combined signature is longer.
Shamir–Tauman (ST) [44]	Pre-compute a trapdoor hash; the online phase signs the hash output with a long-term scheme.	Shorter online signature, but needs a secure trapdoor-hash construction.

TABLE IX: WOTS$^+$ storage and latency for three hash families ($N = 32\,\mathrm{B}$).

w	ℓ	Sig/PK/SK size (B)	Key-gen (µs)				Sign (µs)	Verify (µs)			
			SHA256	BLAKE3	Haraka	Haraka8x		SHA256	BLAKE3	Haraka	Haraka8x
2	265	8,480	34.67	61.17	15.97	9.17	0.86	16.36	31.54	8.04	5.82
4	133	4,256	44.23	87.24	19.08	12.76	0.53	21.33	41.56	9.51	7.08
8	90	2,880	67.22	134.43	27.53	19.16	0.49	61.26	129.82	24.32	18.24
16	67	2,144	105.78	215.34	42.64	30.08	0.35	55.94	128.81	22.53	17.76
32	55	1,760	180.75	361.07	70.82	55.46	0.33	160.33	321.39	67.17	51.15
64	45	1,440	295.13	597.74	117.18	90.26	0.29	279.21	528.91	113.31	86.68

TABLE X: Handshake latency: server-only authentication vs. mutual authentication (µs).

Signature Algorithm	sTLS				mTLS			
	Mean	St. Dev.	P50	P99	Mean	St. Dev.	P50	P99
RSA 2048	2,102.5	351.0	2,091	3,243	2,740.9	284.2	2,721	3,950
ECDSA 256	1,062.6	152.1	1,073	1,484	1,260.9	177.9	1,200	1,645
ED25519	1,077.1	133.3	1,082	1,452	1,225.7	159.3	1,174	1,518
Dilithium II	1,238.5	262.3	1,195	1,978	1,441.3	256.3	1,411	2,187
Falcon 512	1,846.7	179.9	1,784	2,262	2,225.4	176.7	2,207	2,509
SPHINCS$^+$ SHA256-128f-simple	21,804.8	2,930.2	20,961	30,258	43,461.0	4,579.8	43,030	54,165
Looma	**877.3**	**113.3**	**887**	**1235**	**952.2**	**128.5**	**930**	**1266**

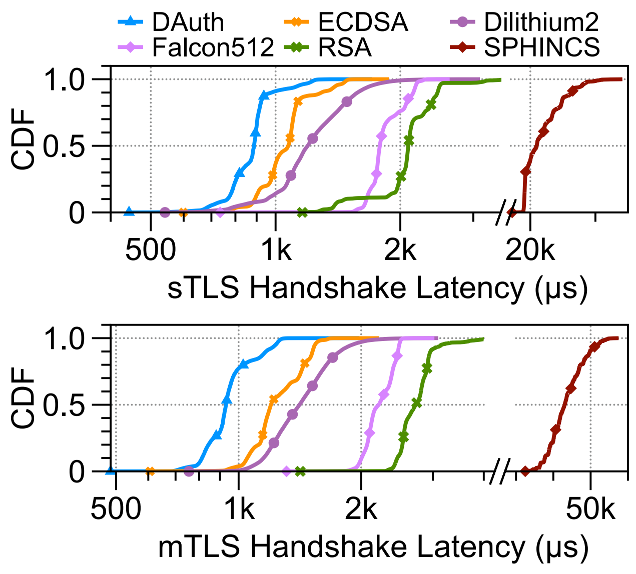


Fig. 10: sTLS (top) and mTLS (bottom) Handshake-latency CDFs.

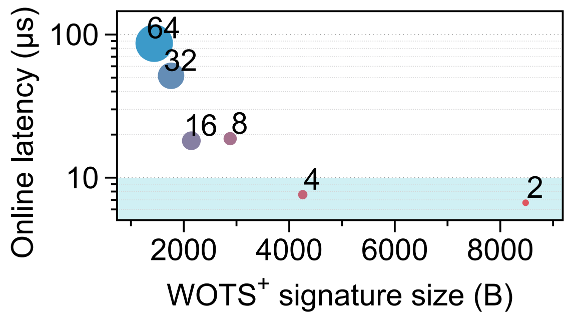
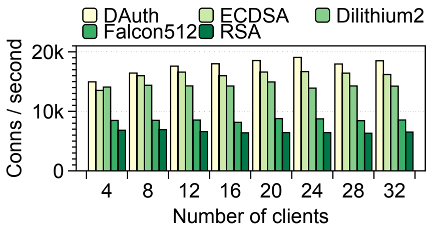


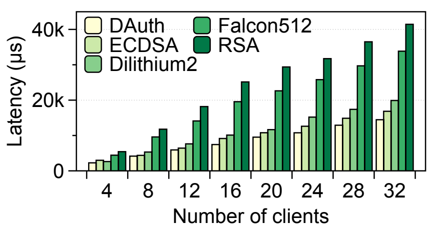
Fig. 11: Signature size v.s. online computation latency of WOTS$^+$ implementation.

TABLE XI: Handshake latency with cache misses (in μs).

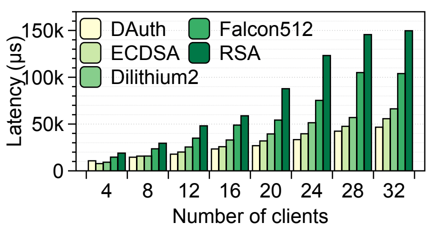
Fallback	Hit/Miss	mTLS				Detail	
		Mean	St. Dev.	P50	P99	Sign	Verify
Hybrid	Hit	952.2	128.5	930	1266	0.39	6.83
	Miss	954.6	128.6	933	1270	0.39	36.38
Dual-sig	Hit	958.7	129.4	933	1273	0.39	6.65
	Miss	958.6	168.3	933	1272	0.39	36.26



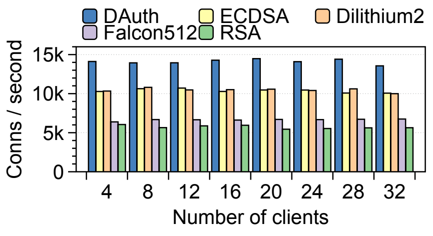
(a) Throughput (sTLS)



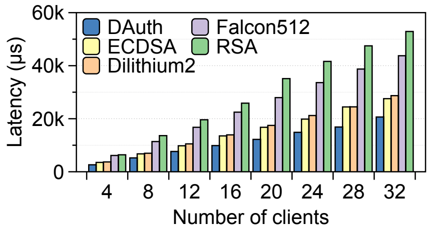
(b) P_{50} latency (sTLS)



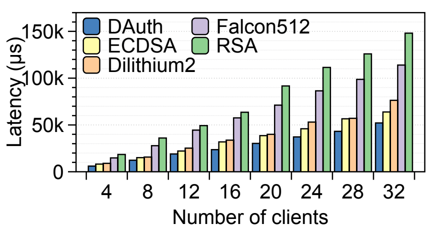
(c) P_{99} latency (sTLS)



(d) Throughput (mTLS)

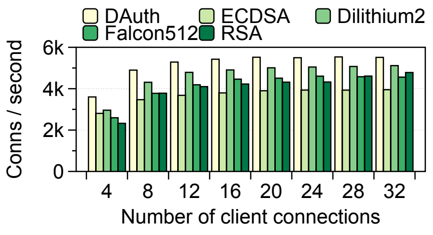


(e) P_{50} latency (mTLS)

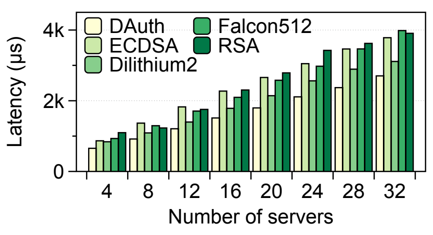


(f) P_{99} latency (mTLS)

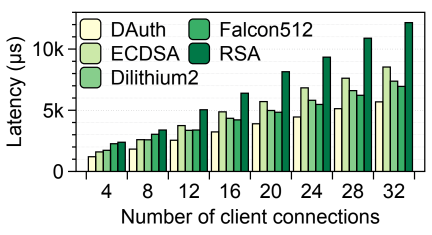
Fig. 12: Handshake throughput over increasing numbers of clients. Latency plots show Looma achieves low latency *while* achieving the highest throughput.



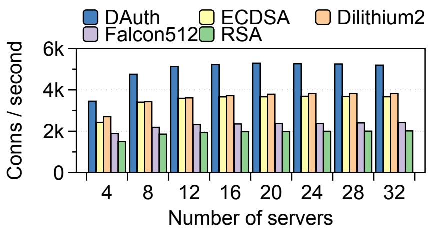
(a) Throughput (sTLS)



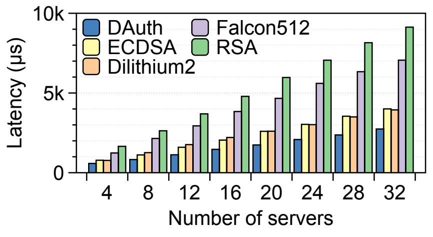
(b) P_{50} latency (sTLS)



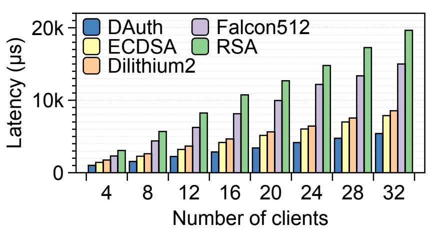
(c) P_{99} latency (sTLS)



(d) Throughput (mTLS)



(e) P_{50} latency (mTLS)



(f) P_{99} latency (mTLS)

Fig. 13: Handshake throughput over increasing number of servers. Latency plots show Looma does not sacrifice latency for high throughput.