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 classical TLS, prompting a shift to post-quantum cryptography. However, post-quantum authentication imposes significant performance overheads, particularly for mutual TLS in cloud environments with high handshake rates. We present Looma, a fast post-quantum authentication architecture that splits authentication into a fast, on-path sign/verify operation and slow, off-path pre-computations performed asynchronously, reducing handshake latency without sacrificing security. Integrated into TLS 1.3, Looma lowers PQTLS handshake latency by up to 44% compared to a Dilithium-2–based baseline. Our results demonstrate the practicality of Looma for scaling post-quantum secure communications in cloud environments.**

I. Introduction

Quantum computers are no longer a mere theoretical possibility. Recent advances from Google, IBM, and Oxford [1], [2], [3], [4] demonstrate steady progress toward scalable, error-corrected quantum hardware, making large quantum machines increasingly plausible. At the same time, widely deployed public-key cryptosystems rely on mathematical problems (e.g., integer factorization) that are intractable for classical computers but efficiently solvable on a sufficiently powerful quantum computer. For example, a recent study from Google shows that a quantum computer with one million logical qubits could break RSA-2048—the most widely used digital signature algorithm—in under a week [5]. The same threat extends to other schemes such as ECDSA and ECDHE, which today underpin authentication and key exchange in TLS.

This looming risk is driving a global shift toward post-quantum cryptography (PQC), a family of algorithms designed to withstand quantum attacks. Since TLS is the backbone of secure communication on the public Internet and in the cloud, there is an urgent need to transition from classical TLS to Post-Quantum TLS (PQTLS). However, PQC schemes generally impose substantially higher computational and bandwidth costs during the handshake than classical algorithms, creating a major deployment barrier, especially at cloud scale.

Modern cloud environments further amplify this challenge. The move from monolithic architectures to microservices and serverless functions means that connections are short-lived and established frequently, with each connection triggering a fresh TLS handshake. 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, further exacerbated by the overheads of post-quantum (PQ) schemes.

Latency is a primary concern for cloud-based applications [6]. While industry leaders such as Amazon [7], Cloudflare [8], [9], and Google [10], [11], [12] have made important progress toward efficient PQTLS—primarily by optimizing key exchange—authentication remains a significant source of high latency, particularly in mTLS deployments.

Motivated by this gap, we introduce Looma[1], a general framework for cloud applications that reduces PQTLS handshake latency by integrating the online/offline signature paradigm into the TLS workflow. Looma replaces the on-path PQ signature with an online/offline variant, splitting authentication into: (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. 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. Looma does not depend on a specific signature algorithm and can be instantiated with different PQ signature schemes (e.g., Dilithium-2 or Falcon); in this paper, we use Dilithium-2 as a representative instantiation.

Our contributions are the following:

1) An in-depth analysis of 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 safely falls back to standard verification when the fast path is unavailable.
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, enabling straightforward integration into Picotls-based applications and protocols.
5) An extensive evaluation showing that Looma consistently outperforms PQ signature algorithms, even classical ones, particularly for mutual authentication in PQTLS.

[1] Low-latency online/offline mutual authentication

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.2026.240074
www.ndss-symposium.org

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 [13], [14], [15], [16]—break applications into many small, frequently interacting services. Each network connection setup typically requires a new TLS handshake, and the ephemeral, autoscaled nature of containers and pods [17] leads to high handshake rates. Because instances are frequently created, destroyed, and rescheduled, techniques like TLS session resumption are often ineffective due to limited state reuse.

Second, **service mesh deployments** (e.g., Istio [18], Linkerd [19], ServiceRouter [20]) and the sidecar pattern introduce extra proxy hops that terminate and re-originate mTLS between adjacent proxies, increasing the number of TLS connections (and thus handshakes) along an end-to-end path.

Third, latency-sensitive, **short-lived flows** dominate inter-service communication in the cloud [21], [22]. Many connections transfer application data for only a small fraction of their lifetime, so handshake latency accounts for a substantial portion of their total time.

Finally, modern datacenter networks are engineered for **low fabric latency** (e.g., $< 50\mu$s) and minimal buffering [23], [24], shifting bottlenecks to the hosts where cryptographic operations, especially PQ handshakes, can dominate application delay and resource usage.

In short, the scale, churn, and short-flow patterns of cloud-native systems have transformed handshake overhead from a negligible concern on the Internet into a critical source of latency. Fast (PQ)TLS handshake is now essential for efficient 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 [25] leverage GPUs, and Kim et al. [26] propose offloading TLS handshakes to SmartNICs. Canal Mesh [27], 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* due to extra data transfer over the PCIe bus or network. Additionally, these approaches require trusting the offload device and heavily depend on hardware capabilities, thereby reducing their general applicability.

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 [28] and session resumption [29], 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 the DNS server, though this requires stronger trust assumptions [30]. PQ proposals like KEMTLS [31] 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, but requiring one additional RTT. However, as highlighted in § II-A, the main concern in modern datacenters is not network fabric latency, but the processing delay of cryptographic operations at the hosts. 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 [32]. In the PQ setting, highly optimized implementations of SNTRUP761 have significantly improved key exchange performance in `OpenSSL` [33]. Notably, most of these works focus on accelerating key exchange.

Orthogonal to these approaches, Looma targets handshake acceleration by decoupling expensive PQ 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) mTLS: TLS is the de facto protocol for end-to-end secure channels on the public Internet and in modern cloud infrastructures. While typical Internet connections authenticate only the server, cloud deployments often require mutual TLS. Mutual authentication (1) blocks unauthorized access via strict identity verification, (2) eliminates man-in-the-middle opportunities during key exchange, and (3) ensures endpoints connect only to legitimate peers, which is critical for sensitive in-boundary data and service interactions.

The mTLS authentication process in Figure 1 uses X.509 certificates and digital signatures to establish trust during the TLS handshake. The `Server Cert` and `Client Cert` messages carry certificates that bind each endpoint's public key to its identity via signatures from a trusted Certificate Authority (CA), whose public key is pre-installed in the peer's trust store. The peer verifies these certificates using the CA's public key, then authenticates the `CertVerify` message by using the public key from the validated certificate to check the signature over the handshake transcript.

The orange-highlighted messages in Figure 1 denote the three extra messages required in mTLS compared to server-side authenticated TLS (sTLS[2]). The server sends a `CertRequest` message to request client authentication. All

[2]We use sTLS to denote server-only authentication and mTLS to denote mutual authentication throughout the paper.

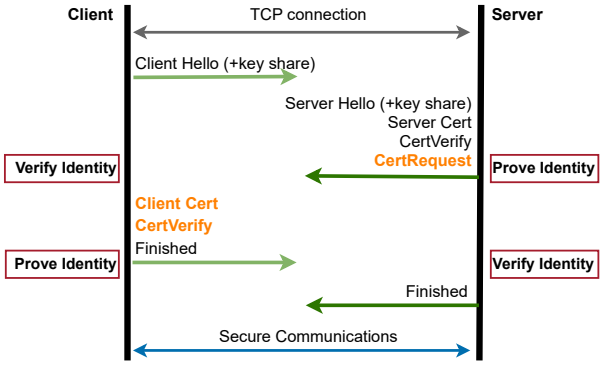


Fig. 1: Mutual authentication in mTLS handshake.

TABLE I: Client-side and server-side sTLS and mTLS handshake components. (•: performed, ○: not performed)

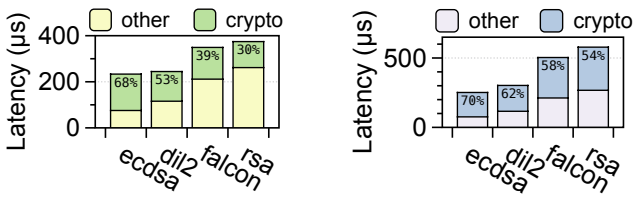
Handshake	Keygen+ex	Verify cert	Sign	Verify
$sTLS_c$	•	•	○	•
$sTLS_s$	•	○	•	○
$mTLS_{c,s}$	•	•	•	•

certificates are assumed to be pre-issued before the handshake begins, so constructing certificate messages does not incur additional asymmetric cryptographic cost during the handshake. However, the client must perform an extra signing operation to produce the `CertVerify` message. The server, in turn, incurs two extra verification operations: one to validate the `Client Cert` and another to verify the `CertVerify` signature. Table I summarizes the main cryptographic operations in sTLS and mTLS at both sides.

b) Cost Breakdown: While mutual authentication strengthens security, it introduces non-trivial computational overhead. As Figure 2 shows, cryptographic operations account for 30%–68% of total sTLS handshake latency and 54%–70% of mTLS latency across different signature schemes, with asymmetric operations (outlined in Table II) dominating the cost. These measurements are taken locally (no network) to isolate CPU overheads, and we use the same key exchange scheme across all tests to highlight the authentication cost of different signature schemes. The high fraction of asymmetric operations in the mTLS handshake indicates that accelerating these operations is an effective means for reducing end-to-end mTLS handshake latency.

In mTLS, the additional client-side *Sign* operation further shifts total handshake time towards signing latency. For example, ECDSA's lowest signing latency (14.9 µs) makes it the most efficient option among the schemes we evaluate. Dilithium-2, despite 47% faster verification than ECDSA, incurs a higher signing cost (51.7 µs, 3.5× slower), further increasing mTLS handshake latency. Falcon-512's 149.2 µs signing pushes total mTLS latency to around 650 µs, i.e., about 80% higher than the 360 µs for ECDSA. Although these differences are on the microsecond scale, they are critical in low-latency datacenters with short-lived connections.

Prior accelerations (§ II-B) do not close this gap. This



(a) sTLS **(b)** mTLS

Fig. 2: Client-side latency incurred by cryptographic and other operations in the handshake process with ECDSA, Dilithium-2, Falcon-512, and RSA-2048 signatures.

TABLE II: Overheads of asymmetric cryptographic operations in TLS handshake. (•: post-quantum, ○: classical)

Operation	Algorithm	Latency (µs)	PQ	Library
Keygen+ex	ECDHE[a]	45.20	○	OpenSSL
	KEM[b]	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

[a] ECDHE is instantiated with secp256r1.
[b] KEM is instantiated with Kyber-512.

motivates Looma, which splits expensive authentication into fast on-path sign/verify, and slow off-path pre-computations, thereby preserving cloud latency targets while enabling quantum resistance during the TLS handshake.

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) [34], Falcon (lattice-based) [35], and SPHINCS$^+$ (hash-based) [36]. 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) [37], [38], 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 typically uses a one-time signature as a building block: highly secure, but limited to signing a single message. The essence is to apply (offline) the ordinary signing algorithm to authenticate 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 pre-signing algorithm 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.[39]—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, PQ signature schemes: SPHINCS$^+$ [40] and XMSS (short for eXtended Merkle Signature Scheme) [41]. 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 3 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 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

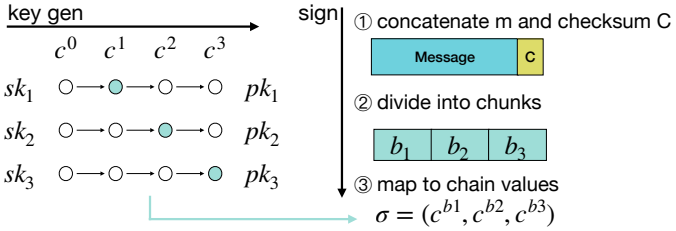


Fig. 3: A toy example of WOTS$^+$ signature scheme: key generation and signing.

$w - 1$ times to each secret-key element. The Winternitz parameter w is tunable, trading signature size against signing/verification cost.

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 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 [42]. Meta's social-media stack is slightly broader (median 6–8), but over 95% of requests still reach fewer than 20 services [42]. 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) [43]. This fixed and limited peer set enables opportunities for optimizations such as pre-computing cryptographic material and 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 services, we enable asynchronous cryptographic preparation and efficient key dissemination. 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 standard primitives remain sound: AEAD record protection provides confidentiality and ciphertext integrity; hash functions are collision resistant; and digital signatures are existentially unforgeable.

Any individual microservice may be taken over through remote-code-execution flaws, container escape, or insider misuse. A compromised service 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 [44]: 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 (i.e., service discovery, the internal CA, orchestrators, etc) 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.

B. Design Scope

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. We focus exclusively on TLS-secured communications within the same datacenter and explicitly exclude connections between external clients and cloud applications or gateways. To meet the demands of latency-sensitive workloads, our proposed Looma scheme must be optimized specifically for performance and scalability within this context. We aim to provide an industry-standard security level (classical 128-bit security and NIST Level 1 PQ security) while improving application-level performance.

Other secure communication schemes that operate at different layers or target distinct deployment models are orthogonal. IPsec [46] functions at the network layer, establishing operator-managed host-to-host tunnels within the trust domain. TLS with pre-shared keys (PSK) [47], used for session resumption, reduces cryptographic overhead but sacrifices forward secrecy and does not apply to full handshakes. QUIC [48] builds upon the TLS 1.3 handshake at the transport layer.

C. Design Choice

While online/offline signatures are well-studied in theory, real-world deployments have largely focused on constrained embedded devices [49], [50]. Looma demonstrates how this paradigm can dramatically reduce PQTLS handshake latency and boost aggregate throughput in high-performance cloud environments.

Among the classical frameworks for converting a standard signature scheme into an online/offline variant—namely, the Even–Goldreich–Micali (EGM) construction [37] using one-time signatures, and the Shamir–Tauman (ST) approach [45] relying on trapdoor hashes (see Table III)—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 [51], 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 PQ portfolio and have undergone extensive public scrutiny. The resulting Dilithium-2/WOTS$^+$ combination maintains purely hash-based PQ 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.

V. Looma Design

We present Looma's high-level architecture (§ V-A), emphasizing its low-latency authentication path (§ V-B), which accelerates session 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 sign/verify operations executed in the handshake process.

A. Looma Architecture

Figure 4 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 scheme. 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 start encrypted application data transfers.

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 operations such as rapidly issuing one-time WOTS$^+$ signatures (FastSign) and quickly authenticating peer signatures (FastVerify), and a *background plane*, which is responsible

TABLE III: Online/offline signature frameworks

Framework	Idea	Pros / cons
Even–Goldreich–Micali (EGM) [37]	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) [45]	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.

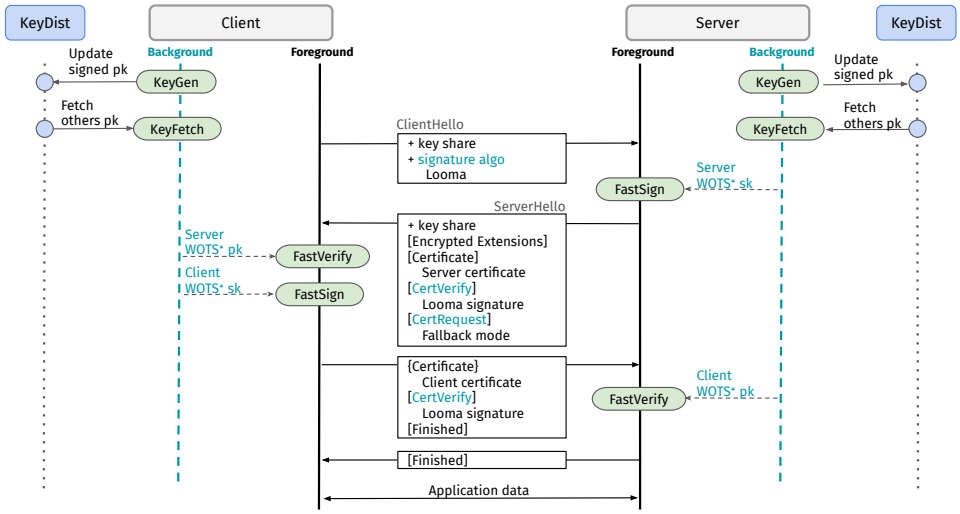


Fig. 4: mTLS 1.3 handshake process with Looma framework for authentication. Dashed green box/text indicates Looma integration.

Fig. 5: Layout of Looma signatures when using dual-sig mode.

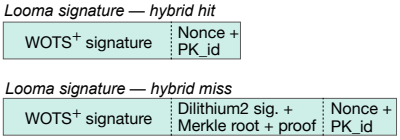


Fig. 6: Layout of Looma signatures when using hybrid mode.

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 PQ 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 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 key fallback scenario: a cache miss, where the verifier enters the handshake without the peer's cached public key. This can happen on the first client–server contact, or when the server fails to refresh a client's key in time. In practice, cache misses are rare on the client side because the client initiates connections and can prefetch the server's public key before starting the handshake. Therefore, cache misses primarily arise in mTLS when the server lacks the client's public key at handshake time.

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 5, each Looma signature in dual-signature mode bundles four main components: (i) a WOTS$^+$ signature on the HT, (ii) a Merkle-tree inclusion proof for the corresponding WOTS$^+$ public key PK, (iii) a Merkle-tree root, and (iv) a Dilithium-2 signature on the Merkle-tree root. The Dilithium public key PK_{d2} used to authenticate the root is carried in client's `Certificate`.

On the server side, verification proceeds in two cases. First, the server checks whether it already holds a cached, pre-verified WOTS$^+$ public key with `PK_id`. If present (cache-hit), it directly verifies the WOTS$^+$ signature against the cached public key using (FastVerify), without re-checking the Merkle tree or the Dilithium-2 signature.

Otherwise (cache miss), the server falls back to full dual-signature verification. It (1) verifies the Dilithium-2 signature on the Merkle-tree root using PK_{d2} from the `Certificate`, (2) recomputes a candidate WOTS$^+$ public key PK^* from the received WOTS$^+$ signature and message, and (3) uses the included Merkle-tree proof to check that PK^*, after compression to a leaf, is consistent with the authenticated Merkle-tree root. If all checks succeed, the server accepts the Looma signature.

This dual-signature strategy guarantees successful authentication even when the public-key cache hit rate is 0%. However, it incurs a fixed per-handshake overhead consisting of three fallback components: the Dilithium-2 signature, the Merkle-tree root, and the Merkle-tree inclusion proof, whose size is $\log_2 B \times 32$ bytes for a binary Merkle tree with B leaves and 32-byte hash values. For instance, a tree with $B = 1024$ leaves yields a proof size of 320 bytes, so the dual-signature fallback components alone contribute roughly 2.4 KB of additional data per handshake. This overhead is incurred even in high cache-hit regimes, where the cached public key would in principle make the fallback path unnecessary, and thus can lead to avoidable bandwidth costs.

Option 2: Bloom-Filter Hybrid Mode. To mitigate the bandwidth overhead of the dual-signature mode, we introduce a more dynamic hybrid strategy. In this mode, each server maintains a Bloom filter [52] over the IDs of peers whose WOTS$^+$ public keys are currently cached. At the start of an mTLS handshake, the server serialises this Bloom filter into a compact bitstring and sends it to the client in a dedicated TLS extension carried by the `CertRequest` message that initiates client authentication requirement.

Upon receiving the `CertRequest`, the client checks membership for its own ID. If the client's ID is not contained in the Bloom filter (a cache miss), it sends a dual-signature as the hybrid-miss Looma signature that includes the WOTS$^+$ signature, the Merkle-tree proof, the Merkle-tree root, a nonce, and a public-key identifier, as depicted in Figure 6. If the client's ID is contained in the Bloom filter (a cache hit), the client instead 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 stable set of peers—a compact Bloom filter (tens of bytes in our settings) easily fits within a single `CertRequest` extension (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 Dilithium-2 authentication. This approach ensures robustness, prevents connection setup failure, and retains performance in the typical, high-cache-hit scenarios. The fallback mode is carried as an extension in the `CertRequest` message (Figure 4).

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-handshake.

Cryptographic security instead stems from the individual endpoints. All public-key materials uploaded to KeyDist are digitally signed by their respective issuers using a standard multi-use, PQ signature scheme (e.g., Dilithium-2), independent of KeyDist. Service 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 over a hash of fields (i)–(iii). The resulting record is uploaded to KeyDist in the form of $\langle \mathsf{KeyUpdate}, \mathsf{keyrecord}, \mathsf{owner\_id} \rangle$ tuple.

Upon receiving the update, 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 resources for frequently accessed keys, thereby reducing the number of key-fetch operations.

3) Key Fetching: Endpoints periodically request updated public keys from KeyDist by sending requests formatted as $\langle \mathsf{KeyFetch}, \mathsf{requester\_id}, \mathsf{owner\_id} \rangle$. Since the endpoint's identity has been 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 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) [53]. 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 PQ 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: $\mathsf{FastVerify}(\mathsf{HT^a}, \sigma^{\mathsf{a}}, \mathsf{PK^a}, \mathsf{r^a}) = 1$. 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 violates the EUF-CMA security property of WOTS$^+$.

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

Hence, any successful forgery must violate either E1 or E2. Since the Dilithium-2 signature scheme is proven EUF-CMA-secure under standard lattice hardness assumptions [34], 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 [39]. In our instance, we use the Haraka hash function [54], which satisfies all these cryptographic properties and has been widely studied in similar PQ signature contexts (e.g., SPHINCS$^+$ [40]). 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. 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 conventional 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 fami-

TABLE IV: 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

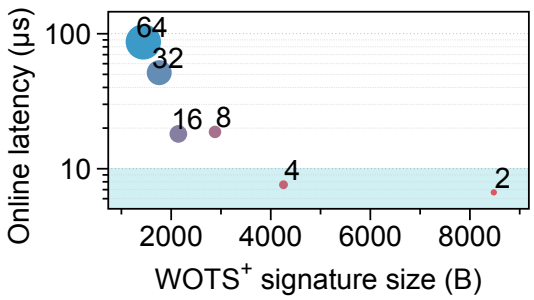


Fig. 7: Signature size v.s. online computation latency of WOTS⁺ implementation.

lies: SHA-256 [55] (from `OpenSSL`), BLAKE3 [56], and Haraka [54]. We found the performance of WOTS⁺ significantly varies 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 1 µs. 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 IV). 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 long messages. Its tree-based design enables highly parallel processing when there is enough data to amortize the overhead, but this additional structure introduces overhead on short, fixed-size inputs, so it performs suboptimally in our setting.

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

Choice of Winternitz Parameter (w). The parameter w controls the trade-off between signature size and computational overhead in WOTS⁺. Larger values of w compress signatures but increase computational cost due to longer hash chains. Conversely, smaller values yield faster computation but larger signatures. As shown in Table IV and Figure 7, increasing from $w=2$ to $w=4$ results in only a modest latency penalty (approximately 1.8 µs) but significantly reduces signature size (by approximately 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 target a Bloom filter configured for up to 15 peers—a threshold that covers more than 99% of bursts in typical microservice deployments while still leaving headroom for infrequent outliers. In a standalone microbenchmark using the Barrust Bloom-filter library [57] with a 56-byte (448-bit) bitmap and $k=20$ non-cryptographic hash probes, we measure an empirical false-positive rate of $p_{obs} \approx 7.6 \times 10^{-4}\%$. The computational cost of each membership check is negligible compared to the rest of the TLS handshake: membership lookups complete in about 280 ns on average (and remain under 1 µs for random lookups), several orders of magnitude below the cost of PQ signature verification and the overall handshake latency.

B. Integration with TLS

To evaluate candidate PQ signature algorithms, we rely on the OQS provider [58] available in `OpenSSL 3.2.0`. To comprehensively evaluate our design, we incorporate classical signatures, PQ candidates, and our own Looma into Picotls [59], 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` code base.

Our implementation is transparent to applications, allowing any existing Picotls-based application to adopt our enhanced

TABLE V: 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	NVIDIA ConnectX-7 / NVIDIA SN3700 100Gbps
Software	Ubuntu 24.04.1 LTS w/ Linux 6.8.0-49-generic kernel

PQ authentication without source code modifications. Picotls is already widely adopted in HTTP/3 deployments [60], [61], offering a realistic testbed for practical evaluation and straightforward extension of our PQ authentication enhancements to modern transport protocols such as QUIC [62].

VIII. Evaluation

We evaluate Looma in comparison to other candidate signature schemes.

Testbed. We use two pairs of identical machines—denoted A2 and I2—with their configurations detailed in Table V. 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 (§ VIII-E). 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)[3]. 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 evaluated Ed25519, but as its performance is comparable to ECDSA, we omit it from the figures. We target 128-bit classical security and NIST Level 1 PQ security. RSA-3072 offers 128-bit classical security, but we show RSA-2048 results due to its wide deployment and faster signature operations.[4] Classic schemes like RSA and ECDSA/Ed25519 would offer 0 bits of security in a PQ 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 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 certificates to avoid CRL or OCSP overhead.

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

[4]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.

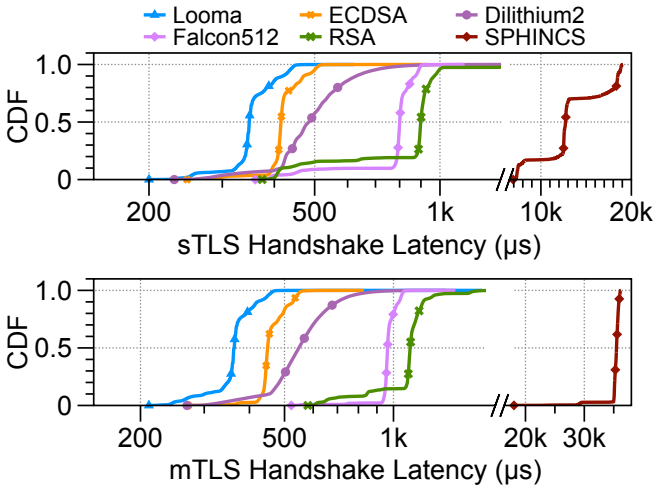


Fig. 8: sTLS (top) and mTLS (bottom) handshake-latency CDFs. (Summary in Table VI).

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 [63]. 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 connection establishments. To mitigate cache-warm effects, we precede the main loop with a client-side warm-up phase and randomize the execution order of algorithms in each trial: for every trial, we shuffle the list of candidate schemes and run the batch for each in that randomized order.

Figure 8 plots the cumulative distribution function of handshake latencies, and Table VI 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. While Falcon-512 and RSA incur higher latencies in both modes, SPHINCS⁺ is markedly slower. We therefore omit SPHINCS⁺ from the comparisons in § VIII-C and § VIII-D.

Results in the I2 setup exhibit a similar trend, with all schemes experiencing approximately 2.5× higher handshake latencies (see Figure 12 and Table X in Appendix A).

B. Authentication Latency

To confirm 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 VII. Across all schemes, Intel-based systems exhibit approximately 1.2–1.5× higher per-operation times than

TABLE VI: 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**

TABLE VII: 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**

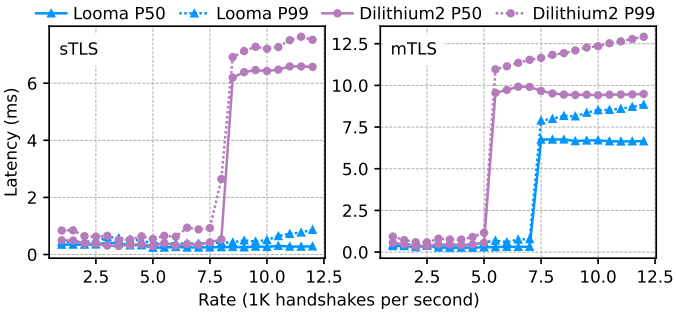


Fig. 9: Handshake latency over increasing request rates for sTLS (left) and mTLS (right).

AMD-based systems, indicating the impact 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 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 backlogs 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 9 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 8k requests per second (RPS) 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.

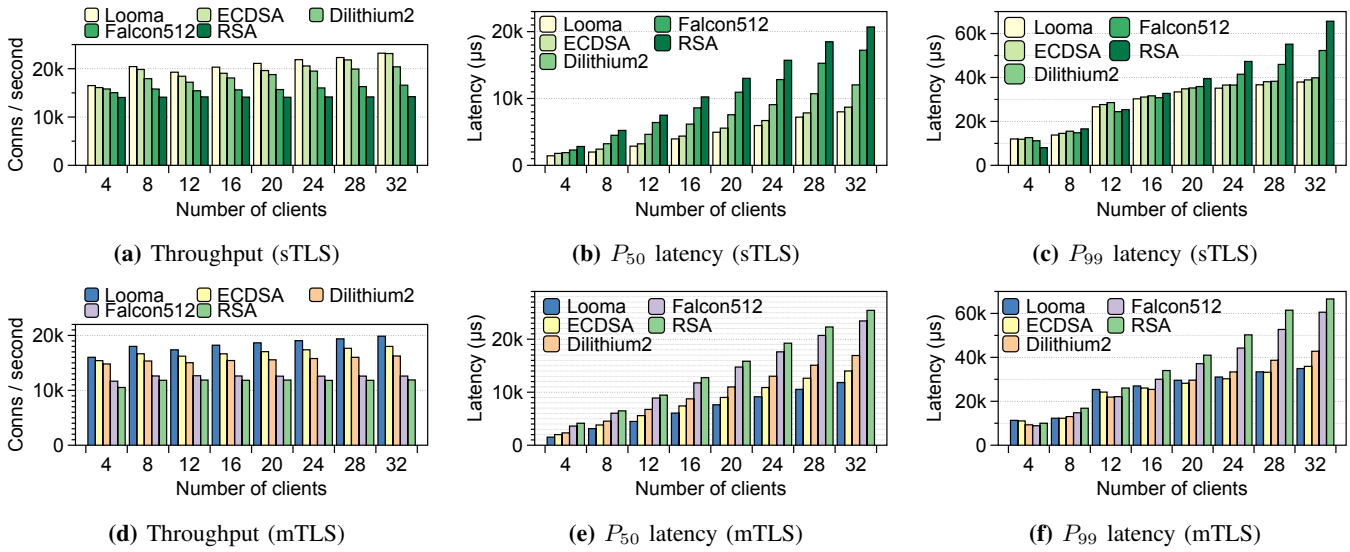
D. Peak Throughput

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 handshake completion. Figure 10 (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 13 in Appendix. In both sTLS and mTLS, the server is nearly saturated at 8 client threads, as throughput is almost reaching its maximum. Beyond this point, additional client threads only increase latency, as requests accumulate in the backlog.

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. Overall, 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 % (A2) and 21 % to 32 % (I2), 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.



(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. 10: Many-to-one: handshake throughput and latency as the number of concurrent clients increases. Looma maintains low latency while delivering the highest throughput.

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

2) One-to-Many Scenario: Next, we evaluate a scenario where a single client concurrently authenticates with many servers. To emulate this, the client runs a single thread, while each of 32 server threads accepts connections. The client then increases its number of simultaneous TCP/TLS handshakes, creating a backlog on the client side, since handshake requests are distributed across multiple server threads.

a) sTLS: Figure 11 (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 VII).

b) mTLS: Figure 11 (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 PQ signing is significantly more expensive than verification (see Table VII). Looma 's online/offline optimization strategy 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.

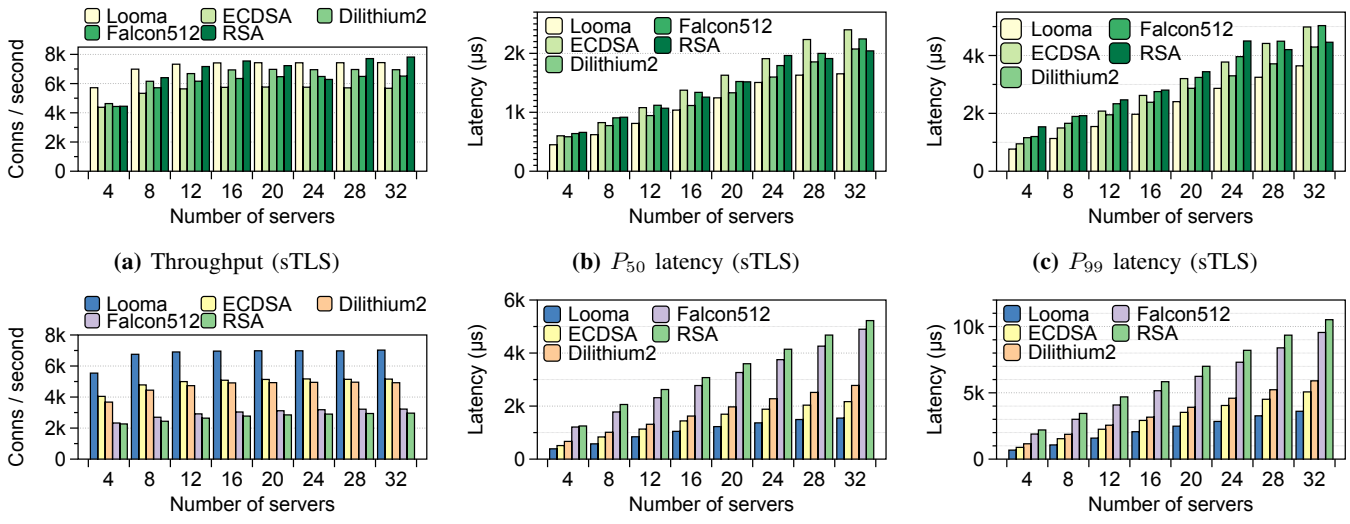
E. Key-Provisioning Overhead

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 [64] to serve these files and evaluate its provisioning capacity by issuing requests from another machine over 128 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. 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 VI), 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 approximately $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 efficiently—about 41,000 key pairs per second on a single dedicated CPU core with our Looma configuration (§ VII-A). 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 23.8 μs per key pair on average, which aligns perfectly with the results in Table IV. There, WOTS$^+$ key generation is the dominant contributor to latency, while the costs of tree



(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. 11: One-to-many: handshake throughput and latency as the number of concurrent servers increases. Looma sustains low latency even while scaling throughput.

construction and root signing are spread across many keys. The offline artifacts occupy (≈17 MB) in total.

b) KeyFetch: Refreshing key records from KeyDist requires one Dilithium-2 verification plus Merkle tree construction over 1024 leaves, for a total of 0.33 ms. After validation, the 1024 public keys (≈4.16 MB) are retained.

c) Impact: Both tasks run off-path. The only visible delay occurs during initial instance bootstrap (≈ 24 ms), which is negligible compared to typical cloud-service startup times. The memory footprints of FastSign (17 MB) and FastVerify (4 MB) together represent under 2% of a 1 GB RAM allocation and under 1% of a 2 GB container—acceptable overhead in modern cloud deployments. In modern cloud infrastructures with multi-gigabit network fabrics, network bandwidth costs are acceptable.

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 VIII reports the handshake latency for cache-hit and -miss cases in A2.

To understand this, we microbenchmark the signing and verification costs and confirm 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 VII). Note that both cache-hit and cache-miss cases benefit from the fast signing time enabled by WOTS$^+$ (≈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.

TABLE VIII: Handshake latency with cache misses (in µs). Hybrid-Hit case is identical to the mTLS case in Table VI

		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

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. Hybrid PQTLS [65], [66], [67] is a transitional approach that combines classical ECDHE with PQ KEMs in TLS 1.3 key exchange, while retaining flexibility in the authentication phase (e.g., using classical or PQ signatures). Looma can substitute traditional signature algorithms such as RSA/ECDSA or PQ signatures (e.g., Dilithium, Falcon) with its online/offline variant. With appropriate configuration, Looma integrates seamlessly with hybrid PQTLS, reducing authentication overhead in both classical and PQ settings. However, Looma is not applicable to KEMTLS [31], 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 sTLS and mTLS over TCP, its benefits extend to other systems

that retain a TLS-style handshake. For example, datapath-encryption frameworks such as Google's PSP [68], kernel-TLS (kTLS) [69], and SMT [70] all perform a TLS 1.3–style handshake in user space; replacing their authentication 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—potentially adopted in serverless platforms for faster function invocation [71]—eliminates one RTT compared to TLS/TCP, so per-handshake signing overhead is highlighted. 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. For example, SmartNICs [26] or remote TLS proxies with special purpose CPU cores [27] 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 [72], 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 called kTLS [73]. Cisco/Cilium uses it for their network observability service [74], [75]. NVIDIA NICs support offloading kTLS processing in the NIC [76].

b) Datacenter-friendly signature: The recently proposed DSig [77] introduces a µs-scale hybrid signature implementation and apply it in datacenter distributed systems that manage frequent transactions and prioritize auditability, such as Byzantine fault-tolerant 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 [78] 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 overhead. Waters et al. [79] propose outsourcing and pre-computing 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 [80] 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 quantum resistance, this paper explored a new approach to fast TLS handshake. Based on the observation that the PQ signature scheme exhibits costly sign and cheap verify performance characteristics, we designed, implemented and evaluated a new digital signature architecture, Looma. The implementation of Looma and benchmark tools used in this paper are available at https://github.com/uoenoplab/looma.

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] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

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

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

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

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

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

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

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

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

[15] 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.

[16] 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.

[17] 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.

[18] Istio, "istio/istio: Connect, secure, control, and observe services," https://github.com/istio/istio.

[19] Linkerd, "linkerd/linkerd2: Ultralight, security-first service mesh for kubernetes," https://github.com/linkerd/linkerd2.

[20] 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.

[21] 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.

[22] 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.

[23] D. Gibson, H. Hariharan, E. Lance, M. McLaren, B. Montazeri, A. Singh, S. Wang, H. M. G. Wassel, Z. Wu, S. Yoo, R. Balasubramanian, P. Chandra, M. Cutforth, P. Cuy, D. Decotigny, R. Gautam, A. Iriza, M. M. K. Martin, R. Roy, Z. Shen, M. Tan, Y. Tang, M. Wong-Chan, J. Zbiciak, and A. Vahdat, "Aquila: A unified, low-latency fabric for datacenter networks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 1249–1266.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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

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

[43] 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.

[44] 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.

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

[46] 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

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

[48] 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

[49] A. C.-C. Yao and Y. Zhao, "Online/offline signatures for low-power devices," *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 2, pp. 283–294, 2012.

[50] J. K. Liu, J. Baek, J. Zhou, Y. Yang, and J. W. Wong, "Efficient online/offline identity-based signature for wireless sensor network," *International Journal of Information Security*, vol. 9, pp. 287–296, 2010.

[51] 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

[52] 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.

[53] 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.

[54] 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.
[55] W. Penard and T. Van Werkhoven, "On the secure hash algorithm family," *Cryptography in context*, pp. 1–18, 2008.
[56] 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/
[57] Tyler Barrus, "Bloom filter implementation written in c," https://github.com/barrust/bloom.
[58] O. Q. Safe, "Oqs provider," https://github.com/open-quantum-safe/oqs-provider.
[59] G. repo, "H2O-picotls project," https://github.com/h2o/picotls.
[60] H2O, "quickly," https://github.com/h2o/quicly.
[61] Private-octopus, "Picoquic," https://github.com/private-octopus/picoquic.
[62] 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
[63] D. Sikeridis, P. Kampanakis, and M. Devetsikiotis, "Post-quantum authentication in tls 1.3: A performance study," *Cryptology ePrint Archive*, 2020.
[64] I. Sysoev and I. NGINX, "nginx web server," https://nginx.org/, accessed: 27 November 2025.
[65] 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/
[66] 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/
[67] Amazon Web Services, "Using hybrid post-quantum TLS with AWS KMS," https://docs.aws.amazon.com/kms/latest/developerguide/pqtls.html, Nov. 2019.
[68] 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.
[69] D. Watson, "KTLS: Linux kernel transport layer security," in *Netdev 1.2*, Tokyo, Japan, 2016. [Online]. Available: https://netdevconf.info/1.2/papers/ktls.pdf
[70] T. Gao, X. Ma, S. Narreddy, E. Luo, S. W. D. Chien, and M. Honda, "Designing transport-level encryption for datacenter networks," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2026, pp. 1502–1519.
[71] 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.
[72] 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.
[73] "Kernel tls offload," https://www.kernel.org/doc/html/latest/networking/tls-offload.html.
[74] J. Fastabend, "Seamless transparent encryption with bpf and cilium," Linux Plumbers Conference 2019.
[75] D. Borkmann and J. Fastabend, "Combining ktls and bpf for introspection and policy enforcement," Linux Plumbers Conference 2018.
[76] 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.
[77] 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.
[78] 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

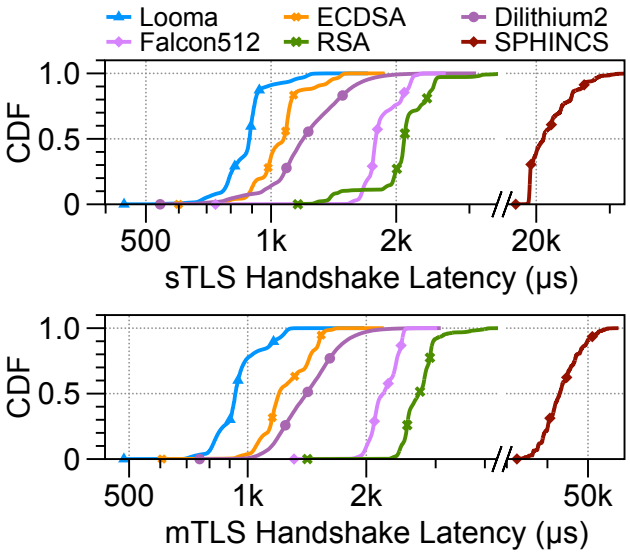


Fig. 12: sTLS (top) and mTLS (bottom) Handshake-latency CDFs on I2.

[79] 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.
[80] 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

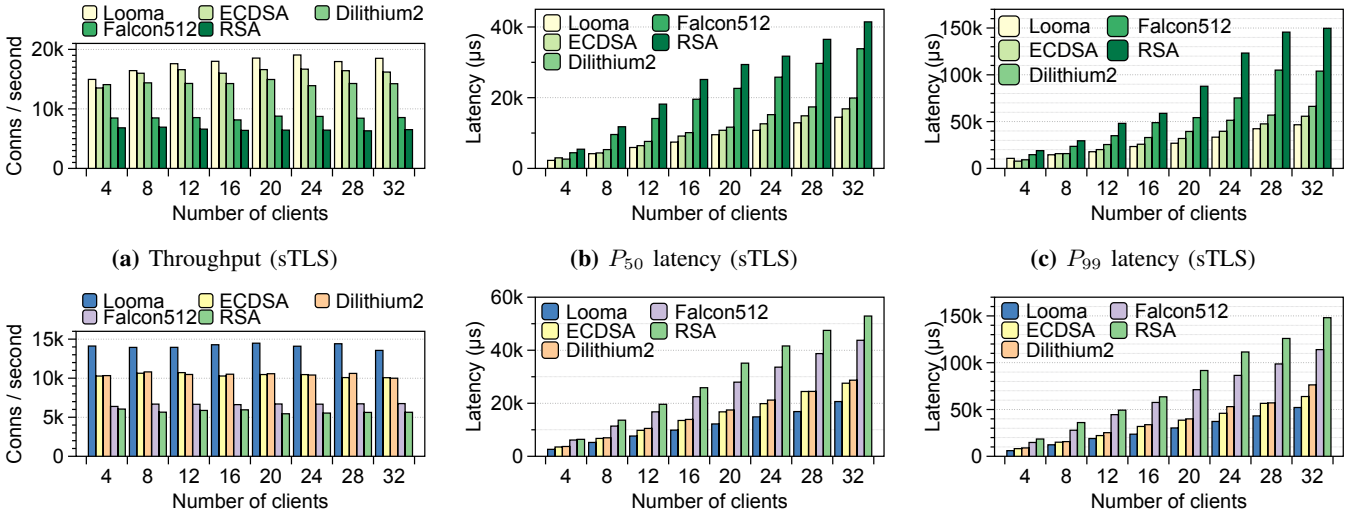
This section presents detailed experimental results from the I2 setup, complementing the A2 results reported in § VII and § VIII. Figure 12 shows the CDFs of sTLS and mTLS handshake latency on I2 (corresponding to Figure 8). Table IX reports the WOTS$^+$ storage and latency trade-offs on I2, as a counterpart to Table IV. Table X summarizes the basic handshake latency on I2, complementing Table VI. Figure 13 and Figure 14 present throughput and latency when scaling the number of clients and servers on I2 (corresponding to Figure 10 and Figure 11). Finally, Table XI shows the end-to-end handshake latency with cache misses on I2.

TABLE IX: WOTS$^+$ storage and latency for three hash families ($N = 32$ B) on I2.

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) on I2.

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**



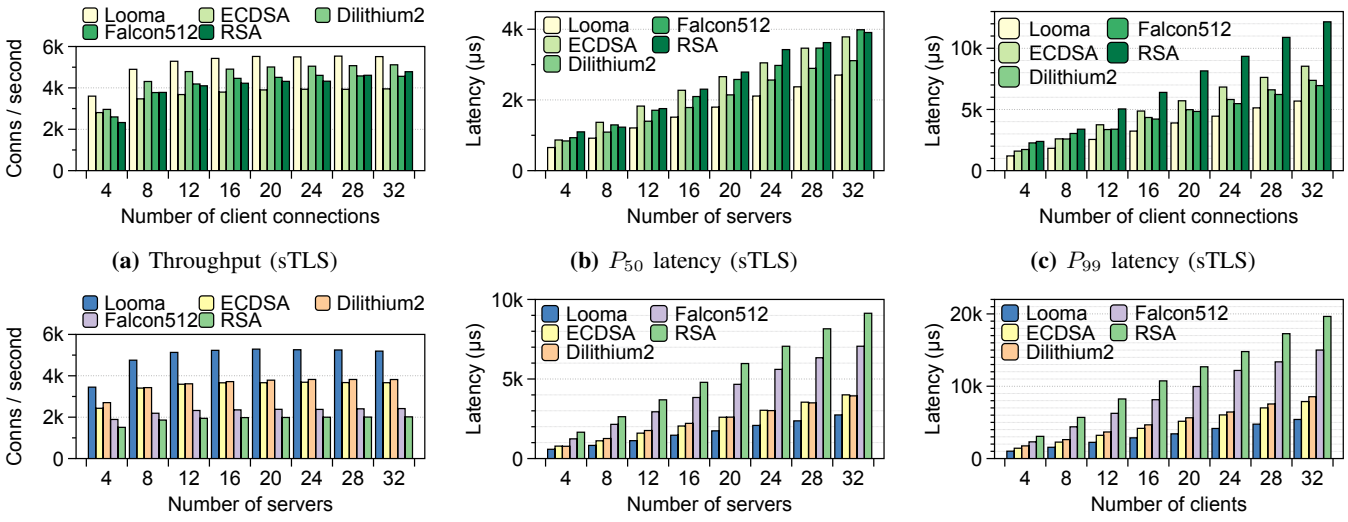
(a) Throughput (sTLS) **(b)** P_{50} latency (sTLS) **(c)** P_{99} latency (sTLS)

(d) Throughput (mTLS) **(e)** P_{50} latency (mTLS) **(f)** P_{99} latency (mTLS)

Fig. 13: Handshake throughput over increasing numbers of clients on I2. Latency plots show Looma achieves low latency *while* achieving the highest throughput.

TABLE XI: Handshake latency with cache misses (in µs) on I2. Hybrid-Hit case is identical to the mTLS case in Table X

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. 14: Handshake throughput over increasing number of servers on I2. Latency plots show Looma does not sacrifice latency for high throughput.