

Post-Quantum Certificate-Based PSK Provisioning for WireGuard via Mutual TLS, DTLS, and QUIC: Design and Performance Study

Che Hang Ng , Emil Künstner , Bastian Küppers , Theodor Schnitzler , and Misha Glazunov 

DACS, Maastricht University, Maastricht, The Netherlands
mikhail.glazunov@maastrichtuniversity.nl

Abstract. We present an architecture that provides WireGuard’s resistance against future quantum adversaries. By preserving the core protocol, our approach minimizes deployment effort and adds certificate-based identities to WireGuard peers. To achieve Post-Quantum (PQ) resistance, we establish an external mutually authenticated control channel built on the TLS 1.3 handshake and layered on top of TCP, UDP, or QUIC. Its key establishment includes Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM). An auxiliary shared secret is derived from the exported keying material of the resulting session and injected into WireGuard’s native Pre-Shared Key (PSK) interface; this derivation is identical for all three transports. To support scalable deployment, the control channel is authenticated with X.509 certificates issued by a Public Key Infrastructure (PKI), using the Module-Lattice-Based Digital Signature Algorithm (ML-DSA) for PQ certificate signing. We develop a Tamarin model of the composition layer and symbolically analyze the secrecy, binding, and provisioning-order properties of the interface between TLS 1.3 handshake and WireGuard, identifying the critical conditions necessary for secure deployment. Finally, we evaluate the design to quantify handshake latency and CPU time for classical, pure PQ and hybrid key establishment, classical and ML-DSA certificates, and the three transports, comparing it against an existing PQ WireGuard baseline achieving state-of-the-art results.

Keywords: WireGuard, PQ-Security, TLS 1.3, Key Establishment, Rosenpass

1 Introduction

Virtual Private Networks (VPNs) are essential for securing online communications. WireGuard [16] has gained prominence as a minimalist, high-performance VPN leveraging the IK handshake pattern from the Noise Protocol Framework [32]. When WireGuard was designed, it deliberately adopted a small, fixed set of modern cryptographic primitives to overcome the problems of earlier TLS versions, where deployments could negotiate many different combinations of cryptographic algorithms. To provide strong security guarantees, WireGuard explicitly chose not to offer protocol and cipher agility to be able to make strong security claims about the algorithms in use. Despite these advantages, we identify three shortcomings in WireGuard:

Absence of a built-in method to provision Pre-Shared Key. WireGuard’s key exchange relies on X25519, an elliptic-curve Diffie-Hellman function based on Curve25519 [8], which is efficient and believed to be secure against classical adversaries. This protection does not extend to a sufficiently capable quantum adversary: Shor’s algorithm [36] solves elliptic-curve discrete logarithms in polynomial time, enabling *harvest-now-decrypt-later* attacks on recorded traffic. WireGuard’s designers anticipated this and, instead of incorporating PQ asymmetric primitives that were impractical at that time, added the option of mixing a PSK into the key derivation during the handshake to strengthen it against future quantum adversaries. This mitigation is only as strong as the PSK and its provisioning. WireGuard, however, deliberately leaves that provisioning out of scope: the key is assumed to be *pre-shared*, meaning that both peers must obtain it beforehand through some external mechanism. If provisioned manually, such a key is typically pairwise and static so a single compromise exposes every session that used it; if transmitted over a channel secured only by classical public-key cryptography, a harvest-now-decrypt-later adversary may eventually recover it as well. WireGuard therefore supplies a mechanism into which post-quantum key material can be injected, but not a post-quantum mechanism for establishing that key material.

Lack of scalable mutual authentication. WireGuard authenticates peers with raw static public keys, which results in a trust model that does not scale. Note that both WireGuard and PKI-based TLS ultimately rest on out-of-band pre-shared trust since a TLS client is shipped with a set of root certificates it did not obtain through any protocol; so the difference is not whether trust is pre-shared but *how*. WireGuard’s trust is pairwise and non-transitive: each peer must directly hold the static public key of every peer it will communicate with, configured out of band before any handshake. For n mutually communicating parties this is $O(n^2)$ trust relationships, each provisioned and rotated by hand, with no way to revoke a key short of editing every configuration that contains it. A PKI restructures the same pre-sharing: each party pre-shares only trust in a small, essentially constant set of certification anchors, and trust in any concrete peer is then derived at connection time by validating a certificate chain back to an anchor. Trust becomes delegated, not pairwise: a signature by a trusted authority vouches for a peer the verifier never configured, so the pre-shared state per participant is constant $O(1)$, membership is open-ended, and revocation is centralized. Previous works have proposed certificate-based authentication for WireGuard, but current extensions lack post-quantum-secure certificates [1].

Lack of verifiable, domain-specific identity. A raw Curve25519 key is self-referential: it identifies only “the party holding the matching private key” and carries no binding to a name, organization, or role, no issuance policy, and no expiry date. A certificate, by contrast, binds the key to a verifiable domain-specific identity under an issuing authority, which is precisely what lets authentication be delegated and revoked as opposed to pairwise and permanent.

In this work, we present an overarching architecture that solves these three shortcomings at once. Rather than modifying the WireGuard protocol itself, this paper follows a composition-based approach. Separating control plane from data plane, we introduce an external, mutually authenticated TLS 1.3 control channel that operates independently of WireGuard and provides certificate-based authentication using an existing PKI. Each

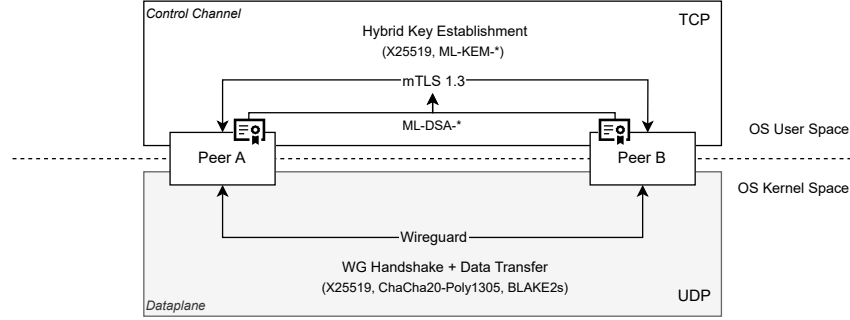


Fig. 1: Overview of the proposed architecture

certificate binds a domain-specific identity (for example, a device or service name within an organization) and the peer’s WireGuard static public key to the certificate’s subject public key, so that a successful control-channel handshake authenticates not merely a key but a named entity authorized to use it. We instantiate this control channel over TLS, DTLS, and QUIC. Its key establishment uses ML-KEM, standalone or in hybrid combination with elliptic-curve Diffie–Hellman, and its certificates are signed with ML-DSA. The exported keying material of the resulting session is injected into WireGuard’s native PSK interface, strengthening WireGuard against quantum adversaries while leaving the protocol, its implementations, and their existing security guarantees untouched. We assume a PKI is already established in which all peers trust the relevant root certificate.

Composing two protocols introduces new security assumptions at their interface and therefore requires formal analysis. We model the interface between the TLS control channel and WireGuard’s PSK in the Tamarin prover, building on the existing WireGuard model of [15], and establish the secrecy, binding, and provisioning-order conditions under which the composition is secure. Figure 1 illustrates the overall architecture. A mutually authenticated TLS 1.3 control plane establishes authenticated and confidential communication between peers. The exported keying material from this plane is injected into a WireGuard PSK, which is subsequently used by an otherwise unmodified WireGuard tunnel. Our contributions are:

- We design a composition that provisions a post-quantum certificate-authenticated PSK into unmodified WireGuard through its existing interface, replacing pairwise key pre-sharing with delegated, PKI-based trust;
- We formally analyze the TLS-to-WireGuard composition layer using the Tamarin Prover and verify secrecy, binding and provisioning-order security properties.
- We implement a proof-of-concept and evaluate the computational and network overhead of the proposed approach across classical, hybrid, and post-quantum cryptographic configurations.

The remainder of this paper is organized as follows. Section 2 reviews related work on PQ VPNs, certificate-based authentication for WireGuard, and PQ TLS. Section 3 presents the architecture, cryptographic design, and formal verification model. Section 4 evaluates

the implementation and verification results. Section 5 discusses security implications, limitations, and future work. Finally, Section 6 concludes the paper.

2 Related Work

NIST has standardized ML-KEM-* (CRYSTALS-Kyber) [6, 13, 31] and selected CRYSTALS-Dilithium (ML-DSA-*) [28], FALCON [17], and SPHINCS+/SLH-DSA [11, 29] for signatures [3], with ML-DSA as the primary recommendation. Code-based KEMs such as Classic McEliece [9, 10] are currently being standardized by ISO. Together with BIKE [5] and HQC [2] these were proposed for a fourth round in the NIST standardization process [4]. SIKE [22, 23] was broken [14], which motivates hybrid constructions [38]. Quantum-safe mechanisms have also been developed outside VPNs, for example for messaging (PQXDH [24]) and for quantum key distribution [25, 37].

Hülsing et al. propose a WireGuard variant that replaces the Diffie-Hellman key-exchange with KEM-based constructions [20]. Lafourcade et al. analyzed their work and constructed an improved version PQ-WireGuard* [26] and combine it with the classical handshake as Hybrid-WireGuard. Hashimoto et. al. propose Rebar a Reinforced KEMs enhanced variant of Hülsing et al. work, utilising smaller packages [18]. The Rosenpass project introduces an extra mechanism leveraging Classic McEliece as an outer layer around the unchanged WireGuard handshake to derive a PQ-secure static key and using Kyber-512 for ephemeral keys and with that the closest to our approach. It utilizes the natively supported option to feed in a pre-shared key (PSK) for the WireGuard handshake [39].

WireGuard lacks certificate-based authentication and relies on the non-PQ-secure X25519 algorithm that utilizes a Diffie-Hellman-key-exchange based on the elliptic curve 25519. Therefore, scalable PQ-secure certificate-based authentication methods are currently researched [1]. The cert-WG protocol extends the handshake to exchange certificates and bind identities to static keys, mitigating identity misbinding attacks [1]. However, modifying the WireGuard protocol comes at the cost of breaking its design simplicity and the authors suggest implementing certification mechanisms around instead of within WireGuard itself.

NetBird¹, Tailscale², Headscale³, and NetMaker⁴ introduce a control channel layered on top of WireGuard rather than altering its protocol internals. This control channel operates "outside" of WireGuard so that it is logically and functionally decoupled from it. Its responsibilities include access control, peer discovery, key distribution, and integration with external identity providers (e.g. OAuth), thereby elevating network management to a higher abstraction level. A key observation, central to both these systems and our design, is the strict separation between the control channel and the data plane. WireGuard, despite performing cryptographic handshakes and key exchanges, is treated purely as a data plane primitive responsible for secure and efficient packet transport between authenticated peers. WireGuard's handshake has been verified in

¹ <https://docs.netbird.io/about-netbird/how-netbird-works>

² <https://tailscale.com/blog/how-tailscale-works>

³ <https://headscale.net/stable/about/features/>

⁴ <https://learn.netmaker.io/>

Tamarin [15, 27] under classical assumptions. We instead model the composition layer between the TLS exporter and WireGuard’s PSK input.

3 Architecture Design

In this section, we explain the fundamental building blocks of our architectural composition and how their properties ensure the security of the overall system. We suggest an auxiliary TLS 1.3 control channel to introduce PQ resistance in WireGuard. This approach ensures backward compatibility by allowing legacy or embedded devices that do not support PQ algorithms, to connect to WireGuard peers that otherwise utilize the control channel for PQ tunnels. The composition between WireGuard and the control channel will then be formalised with Tamarin. Lastly, this section will describe the experimental setup to compare our work with existing work.

3.1 WireGuard

WireGuard is a protocol designed to establish secure, encrypted network tunnels between peers [16]. It uses a compact protocol design and a small cryptographic suite based on well-established primitives, with peers identified by public keys. WireGuard operates at the network layer and can be used to securely route IP traffic between endpoints over an untrusted network. A kernel implementation is available for Linux, eliminating the need for packets to be copied over to userspace, and thus reducing overhead. We provide an overview of all properties of classical WireGuard in Table 4.

Security Properties. WireGuard’s key exchange handshake is 1-Round-Trip-Time (RTT), which minimizes the latency required prior to sending the encrypted messages. Prior to authentication, WireGuard is stateless and does not respond to unauthenticated packets which limits the effectiveness of network scanners finding a peer. Wireguard provides (i) perfect forward secrecy, rendering a compromised long-term static key insufficient to read previously transmitted packets, (ii) a cookie-based mitigation strategy against Denial-of-Service attacks, and (iii) resistance against replay attacks and Key-Compromise Impersonation (KCI).

3.2 Control Channel

For the control channel, we distinguish between three protocols.

TLS 1.3. For the connection and provisioning of the PSK for WireGuard, TLS 1.3 is used [34]. It provides 1-RTT for full handshakes or 0-RTT for session resumption. However, 0-RTT for TLS 1.3 is not used as it requires a PSK (derived from previous handshake or externally obtained), may introduce replay attacks, and early data is not forward secret [34]. As TLS is used solely as a control channel to provision the PSK, the number of RTTs should be as few as possible while still secure to limit the overhead of our solution. TLS 1.3 also provides perfect forward secrecy and protects against downgrade attacks. These properties are important to ensure (old) PSKs used for WireGuard cannot be retrieved even if the TLS keys are compromised in the future, and to ensure an adversary cannot weaken the security of the communication channel. Furthermore, TLS

1.3 can use PQ key-establishment algorithms, which are required for PQ confidentiality and integrity of the transmitted data between a client and server. These include ML-KEM-*, but also hybrid variations of it. Sections 3.6 and 3.7 will motivate the selection of key-establishment groups evaluated in this paper.

The server of the TLS connection can authenticate itself to the client using an X.509 certificate. The client will only proceed with the connection if the certificate provided by the server is valid, i.e. the provided certificate is signed by a common trusted Certificate Authority, has not been revoked, has not expired, and matches the identity of the server [12]. Our proposed method requires mTLS in which both the server and client have to present a valid certificate for mutual authentication [34]. Section 3.3 provides further details about certificate-based authentication and the required fields for our implementation.

Datagram TLS 1.3. The Datagram TLS (DTLS) 1.3 protocol is an alternative option to TLS for applications utilizing UDP as its transport channel [35]. To prevent replay attacks, each DTLS record contains a sequence number. Similarly to WireGuard, DTLS uses stateless cookies to mitigate DoS attacks. Specifically, the use of stateless cookies allows the server to validate the IP address of the client before the server allocates state or performs cryptographic operations, and prevents the server from acting as a packet amplifier for clients with a spoofed IP address [35]. The Maximum Transmission Unit (MTU) is commonly set to 1500 bytes for Ethernet, and IP fragmentation will occur if a packet exceeds this size [35]. For UDP, such packets are not retransmitted automatically, and retransmission has to be initiated by the application or protocol. To prevent IP fragmentation, DTLS handshake messages will be split across multiple DTLS record, with each record fitting into a single UDP datagram. For future sections, mDTLS will refer to DTLS with authentication required from both client and server.

QUIC. QUIC is a transport protocol over UDP that integrates TLS 1.3 for key negotiation to ensure confidentiality and integrity of the transmitted data [21]. As QUIC is based on TLS, QUIC will lose protection against replay attacks if 0-RTT is used as well, but remain to have replay attack protection for 1-RTT connections. QUIC also utilizes Path Maximum Transmission Unit Discovery (PMTUD) to determine the maximum packet size for a given path to avoid IP fragmentation. If PMTUD is not utilized, then QUIC assumes that the given path supports at least 1280 bytes per datagram without IP fragmentation as this is IPv6' minimum MTU. Furthermore, QUIC has built-in protection against amplification attacks by limiting the bytes sent to an unvalidated address, serving the same purpose as stateless cookies in DTLS. An important advantage of QUIC over TLS is the use of Connection IDs (CIDs). Whereas TCP identifies a connection by the IP and port of source and destination, QUIC's use of CIDs allows path migration for established connections without interrupting the connection.

3.3 Authentication and authorization

The certificates used for Mutual TLS, DTLS, and QUIC should adhere to the X.509 Version 3 standard, as our proposed solution requires extensions which earlier versions lack. The digital signatures used to sign the certificates must be PQ-secure as well to prevent an adversary forging certificates and impersonating other identities. Currently, two PQ-secure digital signature algorithms are approved by NIST, ML-DSA [28] and

Stateless Hash-Based Digital Signature Standard (SLH-DSA) [30]. However, as SLH-DSA has a larger combined public key and signature size compared to ML-DSA, it was not considered for our implementation to ensure our approach remains applicable to embedded and on-chip systems. Specific X.509 fields and values are required to be used for the implementation and can be found in Appendix A. Every TLS client who provides a valid certificate to the TLS server is also authorized to establish a WireGuard tunnel. However, this behaviour can be modified to only authorize certificates with pre-defined WireGuard public keys or certificates with specific extensions, to establish a WireGuard tunnel.

3.4 WireGuard PSK provisioning

Keying material can be exported for external purposes based on a TLS handshake [33, 34]. The keying material is derived using HMAC-based Extract-and-Expand Key Derivation Function (HKDF) over the exporter secret, a label, and optionally a context. The exporter secret is established during the TLS handshake. In contrast, the label and context parameters are defined by the application. For our implementation, the label "EXPERIMENTAL-WIREGUARD-PSK" shall be used. The motivation behind this label can be found in Appendix B. Details regarding the context can be found in Section 3.5. A new TLS handshake will be established every 120 seconds to provision a new PSK for WireGuard to ensure forward secrecy. This process is decoupled from WireGuard's rekeying process which establishes new session keys between the peers. Therefore, a newly provisioned PSK will only be used after the next rekeying process of WireGuard.

3.5 Formal Verification

We used Tamarin to symbolically verify the composition layer that derives and installs a WireGuard external PSK from an abstract mutually authenticated TLS 1.3 session. Our verification focuses on the security-critical interface between the two protocols: the binding of TLS identities to WireGuard static public keys, the derivation of a PSK using a TLS exporter, the installation of that PSK into WireGuard, and the subsequent start of WireGuard using the installed PSK. The model addresses the following question: if a WireGuard PSK is derived from a TLS exporter and installed by either endpoint, was it derived from an authorized TLS-authenticated peer relationship, bound to the correct WireGuard static public keys, kept secret from the attacker, and used only after proper installation? Our model proves that once a pair of TLS identities and WireGuard keys is accepted by the composition layer, the resulting exporter-derived PSK is consistently and uniquely bound to that composed tuple.

Model Overview. The Tamarin model represents the composition layer as a sequence of seven main stages:

1. registration of long-term TLS and WireGuard key material;
2. creation of a CA-signed statement binding a TLS identity to a WireGuard static public key;
3. validation of the signed binding statement;
4. construction of a composed peer-binding tuple from the two validated bindings for TLS-identity to WireGuard-key;

5. abstract mutually authenticated TLS completion;
6. derivation and installation of the WireGuard external PSK using a TLS exporter;
7. abstract start of WireGuard after PSK installation.

Each peer is represented by a TLS identity and a WireGuard static public key. In the model, $\$TlsId$ denotes the TLS identity and Wg denotes the associated WireGuard public key. The model abstracts the mechanism that certifies this association as a CA-signed binding statement. This statement may be realized in an implementation either as a separate binding certificate or as information covered by the peer's TLS certificate, for example through a certificate extension.

The signed binding statement contains a domain-separation tag, the TLS identity, and the WireGuard public key. It is accepted only if the signature verifies under a trusted CA public key. Successful validation creates the persistent binding fact:

```
!ValidatedTlsIdBindsWg($TlsId, Wg)
```

This fact records that the composition layer has accepted the association between the TLS identity $\$TlsId$ and the WireGuard public key Wg . The model verifies properties of this validated association, but does not model X.509 parsing, extension processing, revocation, or full certificate-chain validation.

The model enforces uniqueness restrictions on these bindings: one TLS identity may be bound to at most one WireGuard public key, and one WireGuard public key may be bound to at most one TLS identity. It also restricts repeated issuance and validation of the same logical binding statement to avoid multiple witnesses for the same TLS-identity/WireGuard-key pair.

A composed peer binding is then derived from two validated identity/key bindings. We use the suffixes I and R to denote the initiator and responder, respectively:

```
!ValidatedTlsIdBindsWg($TlsIdI, WgI)
!ValidatedTlsIdBindsWg($TlsIdR, WgR)
!ComposedPeerBinding($TlsIdI, $TlsIdR, WgI, WgR)
```

The fact `!ComposedPeerBinding($TlsIdI, $TlsIdR, WgI, WgR)` records that the composition layer is operating over the specific pair of TLS identities and WireGuard public keys. It deliberately does not include a separate administrative peer-authorization policy. The purpose of this model is therefore to verify safe TLS-to-WireGuard composition, not to prove that an external authorization policy admits only intended peer pairs.

The abstract TLS rule models a successful mutually authenticated TLS 1.3 session as producing a fresh exporter master secret (ems) value shared by both peers. This abstraction is conditional on the existence of a composed TLS/WireGuard peer binding. Thus, within the model, TLS completion occurs only for a pair of TLS identities and WireGuard public keys whose bindings have already been validated.

The WireGuard external PSK is derived using a dedicated TLS exporter label and a context containing the TLS identities $\$TlsIdI$ and $\$TlsIdR$, the WireGuard static public keys WgI and WgR , the role labels 'init_role' and 'resp_role', the negotiated algorithms $Algs$, and the composition-version tag 'composition_v1'.

The derived PSK is represented as the result of applying the TLS exporter to the abstract ems, the dedicated exporter label, and this context:

```

tls_exporter(ems,
             'EXPERIMENTAL-WIREGUARD-PSK',
             ctx(...))

```

The model includes an explicit intermediate PSK-derived state before installation, giving the following abstract state progression:

```

TLS complete
-> PSK derived
-> PSK installed
-> WG running

```

This makes explicit that WireGuard can start only after the locally derived PSK has been installed.

3.6 Rosenpass Cryptography Strength

Rosenpass utilises two KEMs, Classic McEliece 460896 and Kyber-512 [39]. The Classic McEliece [9] KEM serves as a static key and is used to authenticate the responder from the initiator. Therefore, the initiator must have access to the public key of the responder. This requires additional storage which can be limited for hardware-embedded or on-chip WireGuard instances and requires another PQ-secure method to transfer this public key. The Kyber KEM is used as an ephemeral key instead.

The NIST SP 800-67 defines five security categories, of which AES-128 is considered to be category one, and AES-192 is considered as category three [7]. According to Varner et al. [39], Classic McEliece 460896 is comparable to 192-bit AES, and Kyber-512 is comparable to 128-bit AES.

As Rosenpass is used as a baseline, a key-establishment algorithm of security category one will be chosen to be comparable to Kyber-512, and a certificate signing algorithm of security category three will be chosen to be comparable to Classic McEliece 460896 for the experiments. These will be ML-KEM-512, the NIST standardised version of Kyber, and ML-DSA-65, respectively, as they share the same NIST security category.

As shown in Table 1, Kyber-512 and ML-KEM-512 have the same key sizes and ciphertext sizes [4, 7, 28, 31]. Classic McEliece 460896 and ML-DSA-65 are used for the same purpose, authentication of the peers. The Classic McEliece keys are a lot larger than the keys for ML-DSA shown in Table 1. This can bring additional network and computational overhead; therefore, for the experiments, this combination of algorithms be compared against each other.

3.7 Experimental Setup

The experimental setup used for Section 4 consists of two Virtual Machines (VMs) running on Debian 13.4.0 on Linux kernel 6.12.74 with Proxmox 9.0.11 as its hypervisor. The hypervisor machine uses a Ryzen 5 5600G and 80GB of DDR4 RAM at 3200 MT/s. For the benchmarks, each VM has one vCPU on ‘host’ preset with 2048MB of RAM assigned with memory ballooning disabled and was done over the local network. Furthermore, for the (m)TLS, (m)DTLS, and (m)QUIC measurements, both VMs were

Table 1: Comparison of Rosenpass cryptographic algorithms and NIST standardized PQ cryptographic algorithms with similar cryptographic strength compared to Rosenpass

Algorithm	Type	NIST security category	Size (bytes)					Signature
			Encaps. key	Decaps. key	Public key	Private key	Cipher-text	
Classic McEliece 460896	KEM	3	524 160	13 608	–	–	188	–
Kyber-512	KEM	1	800	1 632	–	–	768	–
ML-KEM-512	KEM	1	800	1 632	–	–	768	–
ML-DSA-65	Digital signature	3	–	–	1 952	4 032	–	3 309

using WolfSSL⁵ v5.9.1 and ngtcp2⁶ v1.23.0. For WolfSSL and ngtcp2, the specific build flags used can be found in C. For Rosenpass, a modified version based on v0.2.2 was used to include CPU Time measurements within the code. The modified version adds the `cpu_time`⁷ v1.0.0 library to Rosenpass, measures the CPU time from the start of the `event_loop` function until before the PSK is derived, and is built with Nix as a release package. This is equivalent to our TLS, DTLS, and QUIC code, where the export keying material process is not included in the CPU time measurements. To enable connections from clients in TLS and peers in WireGuard, two ports have to be used and forwarded. By default, the TLS port n is user-defined, and WireGuard will then use $n + 1$ as its port. This follows the idea behind port assignments in Rosenpass [39].

The TLS server and client are restarted after every handshake to prevent any remaining state being reused. The server/client handshake secrets are logged to decrypt packets to accurately identify the start of the handshake and the end of the handshake. To eliminate any overhead caused by large packet capture files or secrets logging, these files are rotated every 50 handshakes per key-establishment group and certificate signing algorithm. For TLS, the "ClientHello" message is considered the start of the handshake and the "Finished" message the end of the handshake. For DTLS, the start and end of a handshake is as the "ClientHello" message and "Acknowledgement" message, respectively. For QUIC, the "Initial" packet is considered the start of a handshake and the end of the handshake is considered to be when the server sends the "HANDSHAKE_DONE" frame to the client. The distinction between the three protocols is made based on when both peers are ready to export the keying material.

A 1000 handshakes will be simulated for the mTLS and TLS benchmarks for every pair of key-establishment group (Classical: "SecP256r1", "X25519". Hybrid: "SecP256r1MLKEM768", "X25519MLKEM768". PQ: "ML-KEM-512", "ML-KEM-768", "ML-KEM-1024") and certificate signing algorithm (Classical: "RSA-PSS-2048-SHA256", "RSA-PSS-4096-SHA256", "ECDSA-SecP256r1-SHA256", "Ed25519". PQ: "ML-DSA-44", "ML-DSA-65", "ML-DSA-87"). These key-establishment groups and certificate signing algorithms are chosen to demonstrate the difference between classical algorithms, hybrid algorithms, PQ algorithms, and are used to form a comparison to

⁵ <https://www.wolfssl.com/>

⁶ <https://github.com/ngtcp2/ngtcp2>

⁷ <https://crates.io/crates/cpu-time>

Rosenpass. The same number of handshakes will be simulated for Rosenpass as well. The handshake packets for (m)TLS, (m)DTLS, (m)QUIC, and Rosenpass are captured with TShark 4.2.2. The CPU timings for (m)TLS, (m)DTLS, (m)QUIC are measured with the built-in *clock()* function in C.

3.8 Statistical Analysis

Due to the large number of combinations in protocol, key-establishment groups, and certificate signing algorithms, the statistical analysis will focus on certain key-establishment group and certificate signing algorithm combinations to demonstrate the overhead of transitioning from a classical approach to a hybrid approach and transitioning from a hybrid approach to a full PQ approach. The overhead of the different PQ security strengths and mutual authentication will also be quantified. These combinations will be determined by the experimental results. The slowest and fastest mTLS combination will be compared to their TLS counterpart, respectively. Furthermore, the fastest classical combination will be compared to the fastest hybrid combination to which the fastest PQ combination will be compared to. Furthermore, we will also test for statistical significance between the different PQ parameter sets and NIST security category groups. Lastly, we aim to compare our solution with an existing PQ WireGuard baseline on the same security category as defined in Section 3.6.

The Shapiro-Wilk test will be used to determine if the distribution of the measured latency from the simulations is sampled according to a normal distribution with a 95% confidence level. The Student t-test will be used to determine if the distribution of the two groups is statistically significantly different if the data is not sampled according to a normal distribution. Otherwise, the Mann-Whitney U test will be used instead.

4 Results

To quantify the overhead of the (m)TLS, (m)DTLS, or (m)QUIC handshake, PQ key-establishment groups, PQ certificate signing algorithms, and mutual authentication, 1000 handshakes were simulated. For each algorithm combination, measurements falling outside 1.5-interquartile ranges below the first quartile or above the third quartile are classified as outliers. These outliers are not considered for the statistical analysis of the results in the following sections.

4.1 Baseline Comparison with WireGuard and Rosenpass

With a comparable security category to Kyber-512 and McEliece 460896, the protocols utilized in this work, mTLS, mDTLS, and mQUIC have a lower latency compared to Rosenpass as shown in Figure 2. mDTLS has the lowest latency with 1390.5 microseconds followed by TCP + mTLS with 1461.3 microseconds of latency, which is an increase of 5.1%. mQUIC is compared to TCP + mTLS slower by 50.1% from 1461.3 microseconds to 2193.5 microseconds. However, the difference is the largest and statistically significant between mQUIC and Rosenpass from 2193.5 microseconds to 7381.1 microseconds, corresponding to an increase of 236.5%.

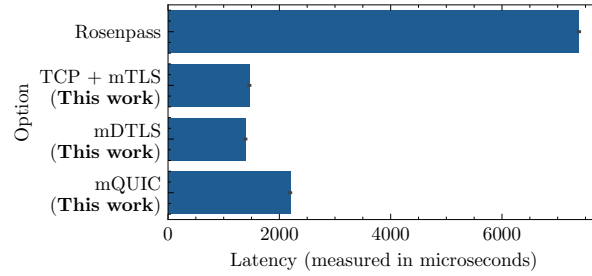


Fig. 2: Comparison of latency between the TCP + mTLS, mDTLS, mQUIC, and Rosenpass, both using full PQ-secure key-establishment groups (ML-KEM-512 with ML-DSA-65 for TCP + mTLS, mDTLS, mQUIC Kyber-512 with Classic McEliece 460896 for Rosenpass)

Table 2: Average number of packets and transmitted bytes required for different protocols, standard deviation is shown within parenthesis. The difference is compared to the baseline (WireGuard). Kyber-512 and Classic McEliece 460896 are used for Rosenpass. ML-KEM-512 and ML-DSA-65 are used for mTLS, mDTLS, and mQUIC.

Protocol	Packets		Bytes	
	Total	Δ	Total	Δ
WireGuard	2 (0)	-	324.0 (0)	-
Rosenpass + WireGuard	6 (0)	+200.0%	2956.0 (0)	+812.3%
TCP + mTLS + WireGuard (This work)	12 (0)	+500.0%	20859.0 (0)	+6338.0%
mDTLS + WireGuard (This work)	25 (0)	+1150.0%	23013.0 (0)	+7002.7%
mQUIC + WireGuard (This work)	33.341 (0.568)	+1567.1%	25545.371 (70.012)	+7784.4%

Additionally, the overhead of the auxiliary handshakes to establish a PQ PSK is compared to WireGuard is shown in Table 2. Rosenpass with WireGuard requires the least number of packets and transmit the least amount of bytes as well. Although, the total number of packets still increased by 200.0% and the total transmitted bytes increased by 812.3% as well. The mTLS, mDTLS, or mQUIC solutions at least double the number of packets required compared to Rosenpass and requires at least 20859.0 bytes which is an increase of 605.6%. The mQUIC solution requires the most number of packets and bytes and has also the highest standard deviation compared to the other protocols.

The CPU time required to finish the handshakes were measured as well with mTLS being the fastest on both client-side and server-side when compared to mDTLS and mQUIC as shown in Figure 3. mDTLS is slower than mTLS by 112.7 microseconds for the client-side which corresponds to 17.7% increase. For the server-side, the CPU time for mDTLS increases by 25.1% and 176.1 microseconds. The difference between mDTLS and mQUIC is larger than mTLS and mDTLS as the client-side CPU time increases by 224.6% relatively and by 1683.3 microseconds in absolute numbers. For the server-side CPU time, the difference increases by only 40.4% or 354.5 microseconds

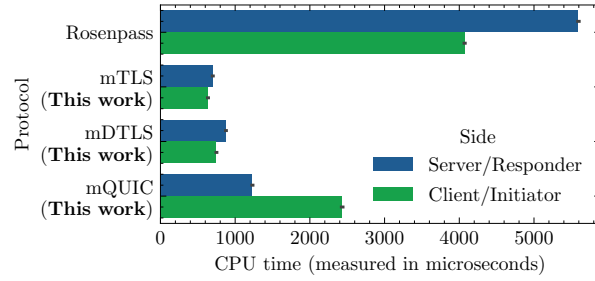


Fig. 3: Comparison of CPU time between the TCP + mTLS, mDTLS, mQUIC, and Rosenpass, both using full PQ-secure key-establishment groups (ML-KEM-512 with ML-DSA-65 for TCP + mTLS, mDTLS, mQUIC Kyber-512 with Classic McEliece 460896 for Rosenpass)

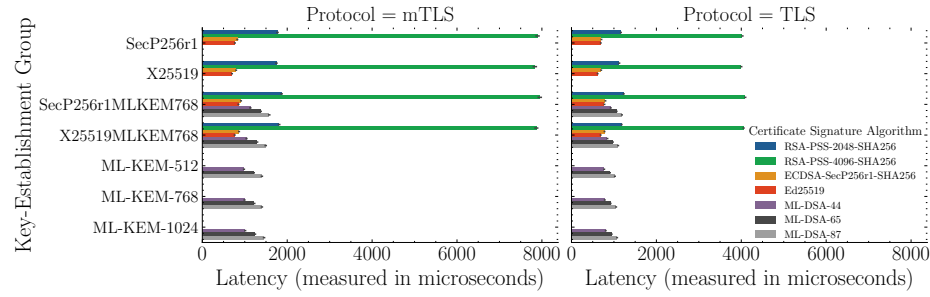


Fig. 4: Latency of (m)TLS 1.3 handshake for different key-establishment groups and certificate-signing algorithms

when switching from mDTLS to mQUIC. If we compare our slowest solution, mQUIC, with Rosenpass, then the CPU time increases by 67.2% or 1638.5 microseconds for the client/initiator side from 2432.7 microseconds to 4071.2 microseconds. The difference in CPU time between mQUIC and Rosenpass is more obvious for the server/responder side as it increases by 354.3% or 4362.8 microseconds from 1231.3 microseconds to 5594.1 microseconds. The measurements are not sampled according to a normal distribution according to the Shapiro-Wilk test and the distributions of the different protocols are significantly different from each other with a p-value less than 0.05 for the Mann-Whitney U test.

4.2 Influence of key-establishment algorithms and certificate signing algorithms

Latency In Figure 4, it can be observed that utilising mTLS leads to a higher latency across every combination of key-establishment groups and certificate signing algorithms. The following paragraphs provides a comparison between the most important key-establishment group and certificate signing algorithm combinations.

Overhead of mutual authentication For the slowest mTLS key-establishment group SecP256r1MLKEM768 and certificate signing algorithm RSA-PSS-4096-SHA256, the mTLS latency overhead is increased by 94.1% from 4095.4 microseconds to 7949.1 microseconds compared to TLS. For the fastest mTLS key-establishment group X25519 and certificate signing algorithm Ed25519, the increase in latency is only 13.7% from 616.5 microseconds to 700.7 microseconds.

Classical to Hybrid approach By switching from the fastest classical combination (X25519 with Ed25519) to the fastest hybrid classical combination (X25519MLKEM768 with ML-DSA-44), the average latency increases from 700.7 microseconds to 1041.4 microseconds with a relative increase of 48.6% for mTLS. However, for TLS, the relative increase is only 37.4% from 616.5 microseconds to 846.9 microseconds.

Hybrid to full PQ approach By moving from the fastest hybrid combination (X25519-MLKEM768 with ML-DSA-44) to the fastest full PQ combination (ML-KEM-512 with ML-DSA-44), the average latency decreases from 1041.4 microseconds to 982.1 microseconds, which is a decrease of 5.7% for mTLS. The same case is observed for TLS with a decrease of 9.0% from 846.9 microseconds to 770.7 microseconds. However, for the full PQ combination, the key-establishment group only achieves NIST security category one, while the PQ part of the hybrid key-establishment group achieves NIST security category three.

Overhead of parameter sets for PQ algorithms Increasing the parameter set of the PQ key-establishment groups from ML-KEM-512 to ML-KEM-1024 while utilizing ML-DSA-44 as the certificate signing algorithm, an increase of 2.9% is observed for mTLS from 982.1 microseconds to 1010.6 microseconds. For TLS, this relative increase is 4.7% instead from 770.7 microseconds to 807.3 microseconds. The overhead of increasing the parameter set for the PQ certificate signing algorithm is larger compared to the overhead of increasing the parameter set for the key-establishment group. The relative increase for mTLS from ML-DSA-44 to ML-DSA-87 is 42.9% from 982.1 microseconds to 1403.5 microseconds, and for TLS 32.7% from 770.7 microseconds to 1022.9 microseconds. These two key-establishment group and certificate signing algorithms combinations can be particularly interesting for use-cases where protection against store-now-decrypt-later attacks is more prominent than PQ-secure authentication or the opposite.

NIST security categories overhead For cases where a certain NIST security category has to be met, such as category 1 (ML-KEM-512 with ML-DSA-44), category 3 (ML-KEM-768 with ML-DSA-65), or category 5 (ML-KEM-1024 with ML-DSA-87), the relative latency overhead is increased by 24.6% from 982.1 microseconds to 1224.1 microseconds and 18.9% from 1224.1 microseconds to 1455.4 microseconds, respectively for mTLS. For TLS, this is an increase of 19.4% from 770.7 microseconds to 920.3 microseconds from NIST category 1 to NIST category 3, and an increase 16.8% from 920.3 microseconds to 1074.7 microseconds from NIST category 3 to NIST category 5.

The overhead of the key-establishment groups, certificate signing algorithms, and mutual authentication can be observed for DTLS and QUIC too in a similar manner compared to TLS. The exact measurements for (m)TLS, (m)DTLS, and (m)QUIC can be found in Appendix E.

CPU Time The CPU time required for the handshake is also dependent on the key-establishment groups and certificate signature algorithms choice as shown in Fig-

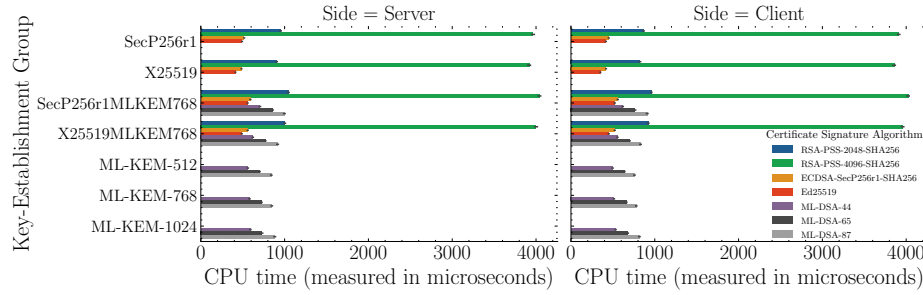


Fig. 5: CPU Time required for mTLS handshake on client-side and server-side with different key-establishment groups and certificate signature algorithms

ure 5. The combination SecP256r1MLKEM768 and RSA-PSS-4096-SHA256 requires the most CPU time on client-side (4023.6 microseconds) and server-side (4040.4 microseconds). However, X25519 and Ed25519 requires the least time for both client-side (350.5 microseconds) and server-side (412.7 microseconds). For the hybrid options, X25519MLKEM768 with ML-DSA-44 at 554.9 microseconds on the client-side and 621.4 microseconds on the server-side. The fastest full PQ combination is ML-KEM-512 paired with ML-DSA-44 at 497.9 microseconds for the client-side and 561.5 microseconds for the server-side. The measurements of the CPU time required for the mDTLS and mQUIC handshakes with different key-establishment groups and certificate signature algorithms can be found in Appendix F alongside the exact measurements for mTLS.

Statistical Analysis The key-establishment groups and certificate signing algorithm combinations shown in Table 3 are selected by the strategy outlined in 3.8 and based on the results in Section 4.2. For every combination, the p-value of the Shapiro-Wilk statistical test is less than 0.05. Thus, the null-hypothesis that the data from the simulation is sampled from a normal distribution can be rejected. Therefore, the Mann-Whitney U test is used to determine if the distributions for the combinations are significantly different from each other. According to the Mann-Whitney U test results, the distribution of the different combinations is not identical as the p-value is less than 0.05 for every combination as well. Hence, the choice of the key-establishment groups, certificate signing algorithm, protocol, and mutual authentication has a significant effect on the measured latency.

4.3 Formal Verification Results

All checked Tamarin lemmas for the composition-only model were successfully verified. The verification results support the intended composition-layer security claims: validated TLS-identity/WireGuard-key bindings are required before PSK installation, installed PSKs agree with prior TLS exporter derivation events, WireGuard starts only after PSK installation, and exporter-derived PSKs remain confidential unless explicitly revealed or compromised according to the model.

The verified properties can be grouped into the following categories.

Table 3: Protocol, ephemeral keys algorithms, and static key algorithms combinations for statistical analysis

Combination One			Combination Two		
Protocol	Key-establish. alg.	Auth. alg.	Protocol	Key-establish. alg.	Auth. alg.
mTLS	SecP256r1MLKEM768	RSA-PSS-4096-SHA256	TLS	SecP256r1MLKEM768	RSA-PSS-4096-SHA256
mTLS	X25519	Ed25519	TLS	X25519	Ed25519
mTLS	X25519	Ed25519	mTLS	X25519MLKEM768	ML-DSA-44
TLS	X25519	Ed25519	TLS	X25519MLKEM768	ML-DSA-44
mTLS	X25519MLKEM768	ML-DSA-44	mTLS	ML-KEM-512	ML-DSA-44
TLS	X25519MLKEM768	ML-DSA-44	TLS	ML-KEM-512	ML-DSA-44
mTLS	ML-KEM-512	ML-DSA-44	mTLS	ML-KEM-1024	ML-DSA-44
TLS	ML-KEM-512	ML-DSA-44	TLS	ML-KEM-1024	ML-DSA-44
mTLS	ML-KEM-512	ML-DSA-44	mTLS	ML-KEM-512	ML-DSA-87
TLS	ML-KEM-512	ML-DSA-44	TLS	ML-KEM-512	ML-DSA-87
mTLS	ML-KEM-512	ML-DSA-44	mTLS	ML-KEM-768	ML-DSA-65
TLS	ML-KEM-512	ML-DSA-44	TLS	ML-KEM-768	ML-DSA-65
mTLS	ML-KEM-768	ML-DSA-65	mTLS	ML-KEM-1024	ML-DSA-87
TLS	ML-KEM-768	ML-DSA-65	TLS	ML-KEM-1024	ML-DSA-87

Executability. The executability lemma confirms that the model admits a complete honest trace, including abstract TLS completion, TLS exporter derivation, PSK installation, and WireGuard start. This ensures that the model is not vacuously secure because of an unreachable protocol flow.

Certificate provenance and binding validation. The certificate-related lemmas establish that every registered TLS-identity/WireGuard-key binding originates from a successfully validated certificate, and that every validated binding certificate was previously issued by the trusted CA abstraction. They also show that a composed peer binding can be derived only when both endpoint bindings have been validated.

Composition precondition for TLS completion. The model verifies that abstract TLS completion occurs only after a composed TLS/WireGuard peer binding has been derived. This connects the abstract TLS session to the validated identity/key bindings used later by the exporter and WireGuard installation steps.

Installation binding correctness. The installation lemmas show that whenever either endpoint installs a WireGuard external PSK, the TLS identities and WireGuard public keys used at installation time correspond to previously registered and validated TLS-ID/WG-key bindings. Thus, installation cannot occur for an unvalidated identity/key tuple.

Exporter and installation agreement. The agreement lemmas show that every PSK installation is preceded by a matching local TLS exporter derivation event for the same role, TLS identities, WireGuard keys, session identifier, algorithm tuple, and PSK value. This verifies that the installed PSK is the one derived by the TLS exporter for the intended context.

WireGuard start ordering. The start-ordering lemmas show that WireGuard can start only after the corresponding PSK has been installed. This establishes the intended lifecycle ordering between PSK derivation, PSK installation, and WireGuard start.

PSK secrecy. The secrecy lemmas show that the exporter-derived PSK is not known to the attacker in traces where the relevant TLS authentication keys have not been corrupted before derivation and the PSK has not been explicitly revealed. This supports the

confidentiality of the TLS-exporter-derived external PSK under the stated compromise model.

Uniqueness and replay resistance. The uniqueness lemmas show that if the same exporter-derived PSK appears in the same local role, then the full bound tuple is equal: the same initiator and responder identities, TLS identities, WireGuard public keys, and algorithm information are involved. The replay-resistance lemmas further show that the same installed PSK for the same local tuple cannot be replayed under a different session identifier, or algorithm tuple.

Overall, the verification supports the claim that the proposed composition layer correctly binds the TLS exporter output to the intended TLS identities, WireGuard static public keys, session context, and protocol roles. The results do not establish correctness of TLS 1.3, correctness of WireGuard, or deployment-specific peer authorization. Instead, they establish that, under the model’s stated abstractions, the PSK is derived based on TLS-exporter and that it is installed and used by WireGuard only as part of a validated and context-bound TLS and WireGuard composition.

5 Discussion and Limitations

5.1 Security Properties of mTLS, mDTLS, mQUIC, and WireGuard

By introducing mTLS 1.3, mDTLS 1.3, or mQUIC with TLS 1.3 to provision WireGuard’s PSK, some security properties of WireGuard are not applicable anymore while some other properties are newly present, as shown in Table 4. The main contribution of utilising the auxiliary control channels is the introduction of PQ resistance to WireGuard tunnels and scalable mutual authentication through the use of certificates. However, this comes at the cost of additional latency as the number of RTTs increases. While WireGuard retains its silent-drop capabilities for unauthenticated packets, the TLS application does not drop any unauthenticated packets and may respond with an alert message instead. On the other hand, by utilising mTLS, the need to (manually) share public keys is eliminated and the identity of a peer is incorporated in a certificate signed by a trusted CA, establishing a chain-of-trust. This does require the identity of the peer to be known and thus the identity hiding property of WireGuard is violated with our approach.

mDTLS and mQUIC servers verify the source IP address to prevent DoS attacks, where state is allocated before reachability of source IP is confirmed, leading to potential CPU- and resource-exhaustion attacks. The underlying mechanisms for these protocols are different; mDTLS utilizes HelloRetryRequest packets, and mQUIC uses Retry packets. This is similar to WireGuard’s cookie mechanism, but WireGuard also allows processing of handshake messages to be deferred in the case of high CPU-load. In contrast, mTLS is dependent on the TCP handshake to verify the reachability of the source IP. mTLS does not allocate state before this is verified. However, TCP itself does allocate state and is susceptible to DoS attacks. As all the auxiliary channels should use their 1-RTT variant, there is no risk for replay attacks introduced.

While mDTLS or mQUIC, have anti-amplification attack protection through a cookie mechanism, or Retry packets respectively, it does not fully negate the risk for amplification attacks. For example, mQUIC only limits the size of responses to at most three

Table 4: Security and networking properties comparison between vanilla WireGuard and WireGuard with our proposed solutions

	Protocols			
Property	WireGuard	WireGuard with PQ-1-RTT-mTLS 1.3 (This work)	WireGuard with PQ-1-RTT-mDTLS 1.3 (This work)	WireGuard with PQ-1-RTT-mQUIC with TLS 1.3 (This work)
Security				
PQ resistance	✗	✓	✓	✓
Silence is a Virtue	✓	✗	✗	✗
DoS Protection	✓	✗	✓	✓
Certificate-based Authentication	✗	✓	✓	✓
Perfect Forward Secrecy (Session Key Secrecy)	✓	✓	✓	✓
Resistance to Replay Attacks	✓	✓	✓	✓
KCI-resistance	✓	✓	✓	✓
UKS-resistance	✓	✓	✓	✓
Initiator Identity Hiding (Active & Passive)	✓	✓	✓	✓
Responder Identity Hiding (Passive only)	✓	✓	✓	✓
Responder Identity Hiding (Active only)	✓	✗	✗	✗
Non-Amplification	✓	✗	✗	✗
Non-Repudiation	✗	✓	✓	✓
Mutual Authentication	✓	✓	✓	✓
Peer Authorization	Configured Peer's Static Keys	Authorization based on CA certs		
Cryptokey routing	Configured Peer's AllowedIPs	Routing based on CA certs		
Networking				
Number of RTT	1-RTT	3-RTT (TCP, TLS, WireGuard)	2-RTT (DTLS, WireGuard)	2-RTT (QUIC, WireGuard)
Single unfragmented UDP datagram	✓	-	✗	✗
Roaming	✓	✗	✗ (Default)	✓

times the size of the bytes received for unvalidated addresses. Also, roaming and path migration is a property only enabled for QUIC by default through the use of CIDs.

Regardless of the auxiliary control channel, the properties for Perfect Forward Secrecy, KCI-resistance, UKS-resistance, and mutual authentications are met. In addition, certificate-based handshake introduces non-repudiation of peer participation which could not have been achieved with a classical WireGuard. Yet, the single unfragmented UDP datagram property is lost, where applicable, for any of the auxiliary control channels. A full overview of the properties of the different protocols can be found in Table 4 with the definitions in Appendix D.

5.2 Handshake latency and network load

For every key-establishment and certificate-signing algorithm combination, the latency for mTLS increased in comparison to TLS, as shown in Figure 4. This increase in latency

is expected as two parties have to send their certificates and verify the certificates instead of one party. The increase in latency by switching from a classical approach to a hybrid approach is due to the hybrid key-establishment group adding ML-KEM-768, and thus adding additional overhead. Moreover, the hybrid approach requires ML-DSA-44 at minimum as WolfSSL does not support hybrid certificate schemes yet. Although it might be counterintuitive, the latency decreases by moving from a hybrid approach to a full PQ approach with ML-KEM-512 and ML-DSA-44. This is due to the hybrid combination utilising ML-KEM-768 (in addition to X25519) instead of ML-KEM-512 for the full PQ combination. Moreover, the additional overhead of generating an X25519 private key and public key, sharing the X25519 public key, and deriving a shared secret from the X25519 keys on both sides is removed for the PQ option as well. Due to the increased parameter sets by using ML-KEM-1024 or ML-DSA-87 instead of ML-KEM-512 or ML-DSA-44, respectively, the increased latency for the increased NIST security category is as expected.

While the difference between TCP + mTLS, mDTLS, and mQUIC are all statistically significant, the absolute difference in latency between mQUIC and TCP + mTLS is larger than the absolute difference of mDTLS compared to TCP + mTLS. We observed two causes for this in our experiments. First, mQUIC includes padding frames leading to more bytes being transmitted and secondly, the packet sizes were limited to 1242 bytes by default to prevent IP fragmentation. However, the maximum packet size observed during the experiments was 1442 bytes for mDTLS and 9806 bytes for mTLS, leading to less packets being sent and thus less overhead of, for example, IP headers or TCP/UDP headers as shown in Table 2.

We also compared the handshake latency of our solution against Rosenpass with algorithms belonging to the same NIST security categories. From our results, we observe that our solution with ML-KEM-512 and ML-DSA-65 is significantly faster than Kyber-512 and McEliece 460896 despite the increased number of packets and total transmitted bytes. As ML-KEM-512 is the NIST standardised version of Kyber-512, the overhead comes from the use of the McEliece algorithm or implementation-wise. The fastest response to the initial packet from the initiator for Rosenpass measured during the experiments takes 3879 microseconds, which is longer than the entire mDTLS, mTLS, or mQUIC handshake. McEliece has also vastly bigger public keys and private keys compared to ML-DSA-65, as previously mentioned in Section 3.6, therefore might not be a suitable choice for embedded systems with limited RAM or storage.

The statistical tests indicate that all combinations in Table 3 are all significant. However, as 1000 handshakes are simulated, even small differences in the distribution may lead to statistical significance. Therefore, it is important to consider effect sizes and determine on a case-by-case basis which algorithms are best suited given the use case.

5.3 Computational overhead

The use of different cryptographic algorithms is primarily compared through handshake latency and CPU time. The CPU time required for the handshake with PQ algorithms is the least for mTLS, followed by mDTLS, and then mQUIC. Although the handshakes are performed by WolfSSL in user-space, mTLS can offload the TCP operations to the Linux kernel. Yet, for mDTLS and mQUIC, which are UDP-based, the user-space process has

to segment the data to ensure there is no IP fragmentation, potentially leading to this increase in CPU time.

Nevertheless, the client CPU time for mQUIC is significantly higher. An hypothesis for this is the additional complexity of QUIC protocol, such as support for CIDs and its mapping to the peer and PMTUD, or additional overhead of our implementation with ngtcp2, but would require further investigation to identify the root cause.

However, computational overhead of the algorithms measured in RAM-utilisation, or memory-bandwidth was not quantified in this study. This is due to the short duration of the handshakes, leading to the need for CPU and/or RAM measurements with high frequency. However, these high-frequency measurements introduced additional overhead, influencing the handshake latency. Future work should therefore investigate other methods, such as memory-bandwidth, to profile the utilization to compare the cryptographic algorithms.

6 Conclusion

We have presented a scalable and deployable design for a composition that provisions a post-quantum certificate-authenticated PSK into unmodified WireGuard through its existing PSK interface. We separate control plane from data plane instead of replacing WireGuard’s handshake. An external, mutually authenticated TLS 1.3 handshake, instantiated over TLS, DTLS, or QUIC, establishes a session using ML-KEM key establishment, standalone or hybrid, and ML-DSA certificates, and the key material exported from that session is injected as WireGuard’s PSK. Because the key is *derived* on both sides from a mutually authenticated handshake rather than transported, it is fresh per session, secret against quantum adversaries, and bound to certificate-authenticated identities. This addresses the three limitations we set out: it supplies the provisioning mechanism on which WireGuard’s own post-quantum resistance depends; it replaces pairwise, manually configured trust with delegated, PKI-based authentication that scales; and it binds each peer’s static key to a verifiable, domain-specific identity.

We modelled the interface between the TLS control channel and WireGuard’s PSK in the Tamarin prover, verifying that a PSK is installed and used only after both endpoints’ certificate-validated bindings are established and that it agrees with the TLS-exporter derivation for its intended context. At a comparable NIST security level, all three transports complete the handshake significantly faster than the Rosenpass baseline (this work lowest at 1390.5 microseconds against a baseline Rosenpass at 7381.1 microseconds); though at a higher byte cost of roughly 21 to 26 kB per handshake against Rosenpass’s 2956 bytes, driven mainly by the ML-DSA certificate chains. Our approach leaves WireGuard’s protocol and security guarantees untouched and applies to any implementation exposing the PSK through its configuration interface.

Acknowledgements This work was partially funded by the ITEA / EUREKA Cluster project PHRESH (Patient Health Response in Emergent and Secure Habitats for Connected Healthcare, project no. 23010).

References

1. Abdelgawad, A., Roy, S.: Incorporating certificates with the wireguard vpn protocol (2022), <https://www.cs.utexas.edu/~dwu4/courses/sp22/project.html>
2. Aguilar-Melchor, C., Blazy, O., Deneuville, J.C., Gaborit, P., Zémor, G.: Efficient encryption from random quasi-cyclic codes. *IEEE Transactions on Information Theory* **64**(5), 3927–3943 (2018). <https://doi.org/10.1109/TIT.2018.2804444>
3. Alagic, G., Alperin-Sheriff, J., Apon, D., Cooper, D., Dang, Q., Kelsey, J., Liu, Y.K., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., Smith-Tone, D.: Status report on the second round of the nist post-quantum cryptography standardization process. NISTIR 8309 (2020), <https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf>
4. Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Liu, Y.K., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., Smith-Tone, D.: Status report on the third round of the nist post-quantum cryptography standardization process (2022). <https://doi.org/10.6028/nist.ir.8413-upd1>
5. Aragon, N., Barreto, P.S.K.M., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Gaborit, P., Ghosh, S., Gueron, S., Güneysu, T., Melchor, C.A., Misoczki, R., Persichetti, E., Richter-Brockmann, J., Sendrier, N., Tillich, J.P., Vasseur, V., Zémor, G.: Bike: Bit flipping key encapsulation (Oct 2024), <https://bikesuite.org/>
6. Avanzi, R., Bos, J.W., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: Crystals-kyber algorithm specifications and supporting documentation (2017), <https://api.semanticscholar.org/CorpusID:198992527>
7. Barker, E., Barker, W.: Recommendation for key management (2025). <https://doi.org/10.6028/nist.sp.800-57pt1r6.ipd>
8. Bernstein, D.J.: Curve25519: New diffie-hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) *Public Key Cryptography – PKC 2006*. Lecture Notes in Computer Science, vol. 3958, pp. 207–228. Springer (2006), <http://cr.yp.to/papers.html#curve25519>
9. Bernstein, D.J., Chou, T., Lange, T., von Maurich, I., Misoczki, R., Niederhagen, R., Persichetti, E., Peters, C., Schwabe, P., Sendrier, N., Szefer, J., Wang, W.: Classic mceliece: Conservative code-based cryptography (2019), <https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/submissions/Classic-McEliece-Round2.zip>, round-2 submission to the NIST PQC project
10. Bernstein, D.J., Chou, T., Cid, C., Gilcher, J., Lange, T., Maram, V., von Maurich, I., Misoczki, R., Persichetti, E., Peters, C., Sendrier, N., Szefer, J., Tjhai, C.J., Tomlinson, M., Wang, W.: Classic mceliece: conservative code-based cryptography: cryptosystem specificationn (Oct 2022), <https://classic.mceliece.org/>
11. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The sphincs+ signature framework. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. p. 2129–2146. CCS ’19, Association for Computing Machinery, New York, NY, USA (2019). <https://doi.org/10.1145/3319535.3363229>
12. Boeyen, S., Santesson, S., Polk, T., Housley, R., Farrell, S., Cooper, D.: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (May 2008). <https://doi.org/10.17487/RFC5280>
13. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehle, D.: Crystals - kyber: A cca-secure module-lattice-based kem. In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. pp. 353–367 (2018). <https://doi.org/10.1109/EuroSP.2018.00032>

14. Castryck, W., Decru, T.: An efficient key recovery attack on SIDH. Cryptology ePrint Archive, Paper 2022/975 (2022), <https://eprint.iacr.org/2022/975>
15. Donenfeld, J.A., Milner, K.: Formal verification of the wireguard protocol (2018)
16. Donenfeld, J.A.: Wireguard: Next generation kernel network tunnel (2017), <https://api.semanticscholar.org/CorpusID:2590070>
17. Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon: Fast-fourier lattice-based compact signatures over ntru (Oct 2020), <https://falcon-sign.info/>
18. Hashimoto, K., Katsumata, S., Niot, G., Wiggers, T.: Revisiting PQ WireGuard: A comprehensive security analysis with a new design using reinforced KEMs. Cryptology ePrint Archive, Paper 2025/1758 (2025), <https://eprint.iacr.org/2025/1758>
19. Housley, R.: Object Identifier Registry for the PKIX Working Group. RFC 7299 (Jul 2014). <https://doi.org/10.17487/RFC7299>
20. Hülsing, A., Ning, K., Schwabe, P., Weber, F., Zimmermann, P.: Post-quantum wireguard. In: O’Conner, L. (ed.) 2021 IEEE Symposium on Security and Privacy (SP). pp. 304–321. IEEE, United States (2021). <https://doi.org/10.1109/SP40001.2021.00030>, 42nd IEEE Symposium on Security and Privacy, SP 2021 ; Conference date: 24-05-2021 Through 27-05-2021
21. Iyengar, J., Thomson, M.: QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000 (May 2021). <https://doi.org/10.17487/RFC9000>
22. Jao, D., Azarderakhsh, R., Campagna, M., Costello, C., Feo, L.D., Hess, B., Hutchinson, A., Jalali, A., Karabina, K., Koziel, B., LaMaccia, B., Longa, P., Naehrig, M., Pereira, G., Renes, J., Soukharev, V., Urbanik, D.: Supersingular isogeny key encapsulation (Sep 2022), <https://sike.org/>
23. Jao, D., De Feo, L.: Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In: Yang, B.Y. (ed.) Post-Quantum Cryptography. pp. 19–34. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
24. Kret, E., Schmidt, R.: The pqxdh key agreement protocol (Jan 2024), <https://signal.org/docs/specifications/pqxdh/>
25. Küstner, H., Dwivedi, A.D., Myrup Pedersen, J.: Kerberos-authenticated classical channel for quantum key distribution: A symmetric-key approach to quantum-safe authentication. In: 2026 IEEE 23rd Consumer Communications & Networking Conference (CCNC). pp. 1–6 (2026). <https://doi.org/10.1109/CCNC65079.2026.11366285>
26. Lafourcade, P., Mahmoud, D., Ruhault, S., Taleb, A.R.: A tale of two worlds: a formal story of wireguard hybridization. In: USENIX Security Symposium (2025), <https://eprint.iacr.org/2025/1179>
27. Meier, S., Schmidt, B., Cremers, C., Basin, D.: The tamarin prover for the symbolic analysis of security protocols. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. pp. 696–701. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
28. NIST: Module-lattice-based digital signature standard (Aug 2024). <https://doi.org/10.6028/nist.fips.204>
29. NIST: Stateless hash-based digital signature standard (Aug 2024). <https://doi.org/10.6028/NIST.FIPS.205>
30. NIST: Stateless hash-based digital signature standard (Aug 2024). <https://doi.org/10.6028/nist.fips.205>
31. NIST, Alagic, G., Dang, Q., Moody, D., Robinson, A., Silberg, H., Smith-Tone, D.: Module-lattice-based key-encapsulation mechanism standard (2024-08-13 04:08:00 2024). <https://doi.org/10.6028/NIST.FIPS.203>
32. Perrin, T.: The noise protocol framework. <https://noiseprotocol.org/noise.html> (2018), [Online; accessed 2025-10-20]

33. Rescorla, E.: Keying Material Exporters for Transport Layer Security (TLS). RFC 5705 (Mar 2010). <https://doi.org/10.17487/RFC5705>
34. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Aug 2018). <https://doi.org/10.17487/RFC8446>
35. Rescorla, E., Tschofenig, H., Modadugu, N.: The Datagram Transport Layer Security (DTLS) Protocol Version 1.3. RFC 9147 (Apr 2022). <https://doi.org/10.17487/RFC9147>
36. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing* **26**(5), 1484–1509 (1997). <https://doi.org/10.1137/S0097539795293172>
37. Spooren, P., Neuhold, A., Ramacher, S., Hühn, T.: Pqc-enhanced qkd networks: A layered approach (2026), <https://arxiv.org/abs/2604.05599>
38. Stebila, D., Fluhrer, S., Gueron, S.: Hybrid key exchange in TLS 1.3. Internet-Draft draft-ietf-tls-hybrid-design-16, Internet Engineering Task Force (Sep 2025), <https://datatracker.ietf.org/doc/draft-ietf-tls-hybrid-design/16/>, work in Progress
39. Varner, K., Lipp, B., Zaeske, W., Schmidt, L., Dua, P.: Rosenpass (white paper). <https://varner25:rosenpass-white-paper.eu/docs/varner25:rosenpass-white-paper-project/whitepaper/> (2025), [Online; accessed 2025-10-20]

Open Science

The artifact repository and reproduction code will be included in the camera-ready version. The repository will contain the source code for establishing an mTLS, mDTLS, and mQUIC connection, derivation of the PSK, and installing the PSK to WireGuard. Helper scripts to generate the X.509 certificates and WireGuard keys will be included too. Finally, a source of our Tamarin verification code.

Ethical Considerations

Generative AI Usage

The authors used generative-AI tools, including ChatGPT, Gemini, and DeepSeek, as assistants for polishing language and restructuring author-written text. With regard to data collection, these tools were used to support the implementation and debugging of measurement, deployment, data-cleaning, and analysis scripts. The tools were not used to generate empirical results, fabricate data, or serve as sources of factual claims or citations.

Human authors made the research-design decisions, ran the experiments, verified the results, artifacts, and source material, reviewed AI-assisted text and code, and take full responsibility for the accuracy, originality, and integrity of the submission.

A X.509 Certificate Field and Values Details

The X.509 certificates used for mutual authentication shall meet the following requirements.

WireGuard's static public key The WireGuard's static public key should be embedded in the X.509 certificate as an ASN.1 UTF8String. To avoid conflicts with registered Object Identifier (OID), the provided implementation uses OID "1.3.6.1.5.5.7.13.1" [19].

Domain-specific identity Any domain-specific identities, such as employee identifiers or names, can be put in the Distinguished Name field.

Subject Alternative Name (SAN) The SAN field of the TLS server's certificate must include the server's IP address or domain name. This is to allow the TLS client to verify that the server it connected to is truly the server the client wants to connect to, mitigating potential Man-in-the-Middle (MiTM) attacks.

Constraints The leaf certificates are intended for authentication. Therefore, they should be disallowed from signing other certificates. Moreover, the certificate for the TLS server must include the "serverAuth" Extended Key Usage (EKU). For the TLS client, the certificate must have the "clientAuth" EKU instead.

B Exporting Keying Material Details

The label chosen for the keying material export process is based on Internet Assigned Numbers Authority's (IANA) recommendations. TLS has pre-defined exporting labels for certain use-cases in the form of "EXPORTER-*", "EXPORTER_*", or "EXPORTER:*". However, as the pre-defined use-cases do not fit our application's need, we defined the label "EXPERIMENTAL-WIREGUARD-PSK" instead. This label is solely meant for this work and is not registered with IANA.

C Artifact Build Configuration

C.1 WolfSSL v5.9.1

```
./configure --enable-milkem --enable-mldsa --enable-dtls --enable-static
--enable-dtls13 --enable-tls-milkem-standalone --enable-curve25519 --
enable-ed25519 --enable-keylog-export --enable-sp --enable-sp-asm --
enable-sp-math-all --enable-intelasm --enable-aesni --enable-aesni-
with-avx --enable-harden --enable-quic --enable-session-ticket --
enable-aesecb --enable-keying-material
```

C.2 ngtcp2 v1.23.0

```
./configure PKG_CONFIG_PATH=/usr/local/lib/pkgconfig --with-wolfssl
```

D WireGuard Security Property Glossary

The exact definitions used for the WireGuard security properties, in comparison to mTLS, mDTLS, or mQUIC are defined as follows [16]

PQ resistance The resulting shared symmetric key should be derived from post-quantum key-establishment algorithms, where both peers are mutually authenticated through post-quantum secure means.

Silence is a Virtue The responder (in the context of WireGuard) or the server (in the context of mTLS, mDTLS, or mQUIC) should not send any responses to unauthenticated packets.

DoS Protection To prevent CPU- and resource-exhaustion attack, the DTLS or QUIC server sends a stateless cookie to the client to verify its source IP address. WireGuard has a stateless cookie mechanism as well, which prevents CPU- and resource-exhaustion attacks too, but also allows the recipient of a message to choose whether to process the initiation or response handshake message or not during high CPU-load. If the recipient of the message chooses not to process the message, then it must provide the sender an cookie which can be used to resend the initial message which the initial recipient has to accept it.

For this work, we consider both CPU- and resource-exhaustion attack, and deferring cryptographic operations due to high CPU-load as DoS protection.

Certificate-based Authentication Authentication should be based on a chain-of-trust certificate construct. The certificate should include the identity of the initiator or recipient in context of WireGuard, or the identity of the client or server in context of mTLS, mDTLS, or mQUIC.

Perfect Forward Secrecy A compromised long-term static key should not be sufficient for a threat actor to decrypt previously transmitted packets. The transport data keys should be derived from ephemeral keying material established during the handshake as well.

Resistance to Replay Attacks Previously sent packets cannot be reused by a threat actor to cause a recipient to process the packet again.

KCI-resistance The protocol should be resistant against attacks where, if a party's long-term static private key is compromised by a threat actor, then the threat actor should not be able to impersonate users to the compromised party.

UKS-resistance The protocol should be resistant against attacks in which two peers A and B derive the same shared secret, while at least one of the two peers believes the shared secret is derived with another party that is not A or B.

Identity Hiding The initiator or recipient (in context of WireGuard), or the client or server (in context of mTLS, mDTLS, or mQUIC) should not provide the other party with its own identity if the party has not authenticated successfully yet.

An active listener can initiate handshakes with the other party (WireGuard Peer or mTLS, DTLS, or mQUIC server) themselves. However, a passive listener can only view packets over the network.

Non-amplification The response should be smaller in size, measured in bytes, compared to the request.

Single unfragmented UDP datagram A message should not exceed the size of one UDP datagram, and thus require the message to be split up into multiple UDP datagrams.

Non-repudiation A peer cannot credibly deny that it has participated in a session or sent a message previously.

Roaming Peers should be uniquely identified with a key, while the external source IP is used to identify the remote endpoint of a peer, and thus allow peers to migrate to another external IPs while still allowing the peer to be identified.

Mutual Authentication Both peers should have successfully verified each other identity when the handshake is finished.

Peer Authorization A peer should only be allowed to finish the handshake successfully if the authorization criteria is met, i.e, the peer's static key is configured on the other peers device for WireGuard, or if the presented certificate of a peer is valid.

Cryptokey routing A peer's data should only be routed according to the cryptokey routing table.

E Supplementary Information - Handshake Latency

E.1 (m)DTLS and (m)QUIC measurements

E.2 Exact Measurements

F Supplementary Information - CPU Time

F.1 mDTLS and mQUIC measurements

F.2 Exact Measurements

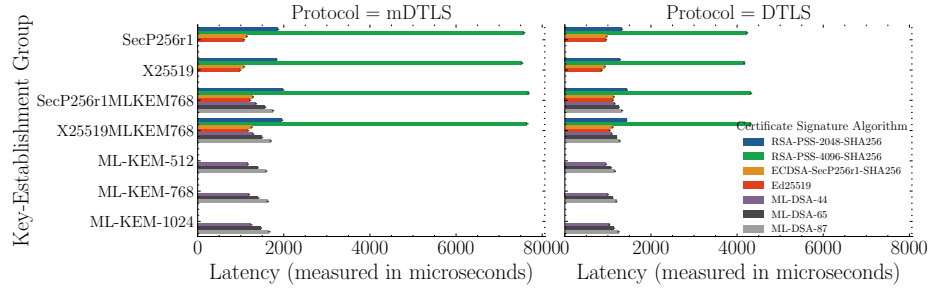


Fig. 6: Latency of mDTLS and DTLS handshake for different key-establishment groups and certificate-signing algorithms

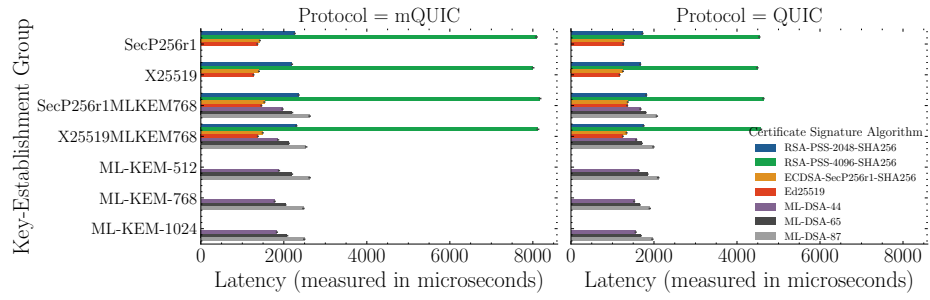


Fig. 7: Latency of mQUIC and QUIC handshake for different key-establishment groups and certificate-signing algorithms

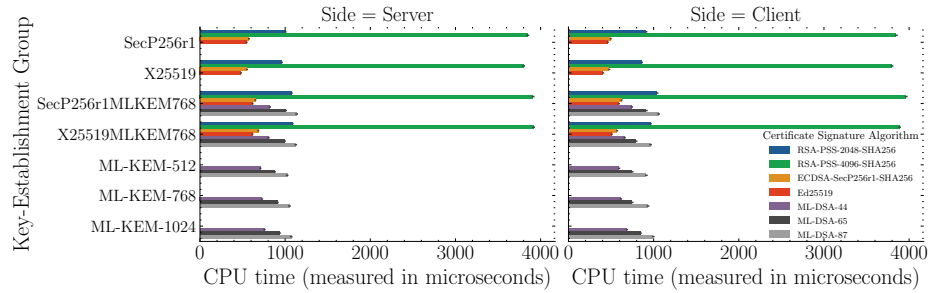


Fig. 8: CPU Time required for mDTLS handshake on client-side and server-side with different key-establishment groups and certificate signature algorithms

Table 5: Latency measurements in microseconds for TLS, mTLS, TCP + mTLS, DTLS, mDTLS, QUIC, and mQUIC with different ephemeral keys algorithms, and static key algorithms combinations. Standard deviation is shown in brackets.

Key-establish. alg.	Auth. alg.	Protocols						
		TLS	mTLS	TCP + mTLS	DTLS	mDTLS	QUIC	mQUIC
SecP256r1	RSA-PSS-2048-SHA256	1165.3 (95.3)	1771.8 (192.8)	2058.0 (202.8)	1325.0 (117.2)	1868.0 (124.0)	1724.7 (121.4)	2263.2 (120.7)
SecP256r1	RSA-PSS-4096-SHA256	4027.9 (141.6)	7897.9 (561.7)	8182.3 (562.9)	4230.1 (163.7)	7576.2 (191.3)	4537.2 (158.8)	8084.4 (241.9)
SecP256r1	ECDSA-SecP256r1-SHA256	706.3 (63.2)	825.0 (91.1)	1075.3 (107.5)	971.0 (107.2)	1141.5 (118.7)	1275.7 (85.2)	1424.2 (90.5)
SecP256r1	Ed25519	690.6 (63.2)	768.1 (88.5)	1035.7 (105.4)	956.7 (108.6)	1078.8 (110.2)	1256.8 (85.6)	1362.4 (89.0)
X25519	RSA-PSS-2048-SHA256	1127.0 (94.0)	1745.0 (190.7)	1996.3 (200.8)	1281.2 (118.9)	1827.5 (121.2)	1665.9 (117.2)	2195.5 (127.0)
X25519	RSA-PSS-4096-SHA256	4001.4 (138.3)	7834.9 (537.9)	8089.8 (548.7)	4164.7 (146.0)	7530.7 (197.7)	4490.2 (168.3)	8005.7 (222.8)
X25519	ECDSA-SecP256r1-SHA256	699.9 (66.6)	797.2 (83.4)	1025.9 (99.8)	931.6 (112.6)	1082.0 (113.2)	1253.5 (84.8)	1401.0 (94.8)
X25519	Ed25519	616.5 (63.2)	700.7 (78.2)	926.0 (93.8)	859.0 (105.0)	991.0 (112.5)	1172.6 (83.7)	1267.9 (84.9)
SecP256r1MLKEM768	RSA-PSS-2048-SHA256	1231.0 (86.1)	1867.1 (205.0)	2225.9 (221.6)	1440.3 (112.0)	1988.4 (124.4)	1811.6 (122.2)	2356.1 (128.0)
SecP256r1MLKEM768	RSA-PSS-4096-SHA256	4095.4 (134.3)	7949.1 (535.9)	8310.8 (548.7)	4322.4 (145.0)	7679.2 (194.4)	4635.7 (176.1)	8175.0 (222.2)
SecP256r1MLKEM768	ECDSA-SecP256r1-SHA256	800.3 (67.1)	911.6 (103.6)	1228.2 (121.9)	1136.3 (111.4)	1284.1 (110.6)	1369.4 (92.0)	1532.2 (99.0)
SecP256r1MLKEM768	Ed25519	777.4 (62.4)	851.6 (89.9)	1188.6 (110.8)	1115.2 (109.7)	1218.8 (109.0)	1353.5 (93.1)	1458.1 (91.2)
SecP256r1MLKEM768	ML-DSA-44	925.6 (81.1)	1131.8 (139.5)	1444.6 (159.7)	1154.8 (110.8)	1353.3 (123.4)	1674.7 (133.0)	1967.5 (141.6)
SecP256r1MLKEM768	ML-DSA-65	1063.1 (90.9)	1375.1 (177.0)	1686.1 (190.3)	1252.2 (116.3)	1566.4 (140.4)	1798.0 (142.5)	2198.5 (162.4)
SecP256r1MLKEM768	ML-DSA-87	1188.5 (92.1)	1577.3 (194.2)	1892.4 (209.8)	1328.5 (122.6)	1750.4 (141.2)	2069.7 (152.5)	2622.2 (174.1)
X25519MLKEM768	RSA-PSS-2048-SHA256	1182.0 (90.2)	1809.1 (200.4)	2129.7 (213.4)	1424.5 (122.1)	1962.0 (122.7)	1749.4 (120.6)	2299.8 (139.3)
X25519MLKEM768	RSA-PSS-4096-SHA256	4054.8 (138.3)	7869.8 (479.7)	8205.1 (506.5)	4308.3 (150.8)	7651.0 (209.8)	4576.6 (170.6)	8121.6 (253.0)
X25519MLKEM768	ECDSA-SecP256r1-SHA256	778.5 (64.5)	866.1 (103.3)	1163.4 (124.4)	1109.8 (110.6)	1253.9 (117.8)	1340.3 (86.7)	1492.4 (90.4)
X25519MLKEM768	Ed25519	687.6 (63.0)	762.8 (90.0)	1059.2 (109.0)	1043.1 (107.4)	1176.9 (114.8)	1249.7 (85.1)	1374.6 (93.9)
X25519MLKEM768	ML-DSA-44	846.9 (82.1)	1041.4 (124.1)	1319.0 (143.0)	1086.9 (107.4)	1290.4 (121.9)	1570.4 (130.0)	1863.0 (126.2)
X25519MLKEM768	ML-DSA-65	975.2 (88.2)	1291.5 (172.9)	1567.6 (188.8)	1192.4 (115.9)	1496.4 (136.5)	1707.2 (139.0)	2118.0 (153.9)
X25519MLKEM768	ML-DSA-87	1100.0 (89.2)	1493.3 (199.2)	1768.6 (213.6)	1274.8 (122.3)	1697.3 (143.4)	1985.4 (164.4)	2538.4 (176.3)
ML-KEM-512	ML-DSA-44	770.7 (82.6)	982.1 (129.3)	1227.1 (145.5)	956.8 (111.9)	1165.3 (127.2)	1630.6 (153.1)	1882.8 (134.0)
ML-KEM-512	ML-DSA-65	906.7 (90.0)	1213.4 (152.4)	1461.3 (167.3)	1071.2 (116.3)	1390.5 (142.7)	1838.6 (195.7)	2193.5 (170.7)
ML-KEM-512	ML-DSA-87	1022.9 (91.4)	1403.5 (173.9)	1651.4 (190.1)	1162.9 (120.1)	1593.2 (147.1)	2102.0 (186.7)	2621.5 (188.3)
ML-KEM-768	ML-DSA-44	784.3 (79.3)	993.2 (124.9)	1250.7 (139.4)	999.5 (114.3)	1193.9 (121.0)	1521.6 (128.3)	1780.5 (128.4)
ML-KEM-768	ML-DSA-65	920.3 (90.0)	1224.1 (159.5)	1486.7 (175.3)	1104.2 (119.0)	1407.6 (134.9)	1654.7 (141.0)	2035.7 (150.5)
ML-KEM-768	ML-DSA-87	1051.5 (93.2)	1404.7 (172.4)	1664.8 (186.0)	1196.4 (122.9)	1628.3 (149.5)	1899.5 (162.7)	2472.5 (165.1)
ML-KEM-1024	ML-DSA-44	807.3 (76.3)	1010.6 (130.2)	1277.1 (144.6)	1031.2 (114.1)	1246.3 (132.9)	1559.6 (124.2)	1832.7 (133.4)
ML-KEM-1024	ML-DSA-65	946.6 (90.1)	1241.3 (162.2)	1510.8 (175.3)	1136.5 (118.4)	1468.3 (141.2)	1681.9 (141.2)	2079.3 (153.9)
ML-KEM-1024	ML-DSA-87	1074.7 (91.3)	1455.4 (184.9)	1725.9 (201.4)	1247.0 (127.5)	1664.4 (144.2)	1962.0 (159.7)	2491.8 (155.9)

Table 6: CPU time measurements in microseconds for mTLS, mDTLS, and mQUIC with different ephemeral keys algorithms, and static key algorithms combinations. Standard deviation is shown in brackets.

Key-establish. alg.	Auth. alg.	CPU Time (microseconds)					
		mTLS		mDTLS		mQUIC	
		Client	Server	Client	Server	Client	Server
SecP256r1	RSA-PSS-2048-SHA256	866.8 (93.3)	952.6 (110.4)	913.3 (64.4)	1002.1 (79.6)	2531.2 (133.0)	1272.9 (104.4)
SecP256r1	RSA-PSS-4096-SHA256	3910.7 (209.0)	3964.2 (227.5)	3849.2 (120.1)	3849.7 (124.8)	8353.2 (237.4)	4184.4 (155.3)
SecP256r1	ECDSA-SecP256r1-SHA256	451.0 (59.0)	516.7 (65.2)	496.3 (39.7)	575.9 (50.4)	1694.3 (105.8)	767.4 (61.5)
SecP256r1	Ed25519	416.7 (57.4)	492.9 (60.5)	462.8 (38.8)	546.6 (45.6)	1644.5 (104.9)	753.3 (64.4)
X25519	RSA-PSS-2048-SHA256	824.1 (99.4)	905.7 (113.1)	862.4 (55.6)	960.2 (74.6)	2438.7 (138.9)	1224.0 (106.8)
X25519	RSA-PSS-4096-SHA256	3856.6 (206.4)	3918.4 (250.4)	3798.5 (112.4)	3800.1 (121.1)	8251.2 (223.0)	4132.2 (154.8)
X25519	ECDSA-SecP256r1-SHA256	418.3 (56.8)	484.8 (56.4)	479.3 (44.8)	552.2 (46.3)	1655.6 (109.7)	752.1 (67.1)
X25519	Ed25519	350.5 (49.8)	412.7 (52.3)	408.1 (39.4)	481.9 (43.1)	1516.4 (99.7)	686.5 (62.0)
SecP256r1MLKEM768	RSA-PSS-2048-SHA256	959.3 (84.7)	1044.0 (138.9)	1043.7 (67.0)	1075.3 (79.2)	2680.7 (141.7)	1361.5 (108.4)
SecP256r1MLKEM768	RSA-PSS-4096-SHA256	4023.6 (215.1)	4040.4 (234.1)	3966.2 (111.7)	3910.4 (126.6)	8502.3 (224.9)	4269.7 (141.7)
SecP256r1MLKEM768	ECDSA-SecP256r1-SHA256	558.5 (74.2)	591.5 (71.0)	622.8 (45.6)	650.3 (49.0)	1849.1 (110.7)	900.0 (76.4)
SecP256r1MLKEM768	Ed25519	522.2 (68.8)	557.9 (69.7)	593.4 (41.4)	617.4 (46.6)	1792.6 (105.3)	877.7 (72.2)
SecP256r1MLKEM768	ML-DSA-44	618.8 (90.2)	706.0 (97.9)	744.7 (74.0)	820.1 (77.4)	2266.4 (150.1)	1193.8 (108.1)
SecP256r1MLKEM768	ML-DSA-65	764.1 (115.5)	854.8 (125.0)	915.2 (100.1)	1005.8 (98.3)	2494.4 (173.6)	1345.3 (120.3)
SecP256r1MLKEM768	ML-DSA-87	909.4 (136.3)	997.0 (143.0)	1058.1 (108.8)	1134.8 (107.8)	2916.8 (180.9)	1586.0 (138.8)
X25519MLKEM768	RSA-PSS-2048-SHA256	923.1 (102.6)	1001.7 (125.8)	964.6 (67.4)	1085.6 (78.3)	2595.1 (148.0)	1327.9 (116.4)
X25519MLKEM768	RSA-PSS-4096-SHA256	3957.9 (201.5)	3999.6 (228.0)	3887.1 (115.8)	3917.2 (125.0)	8419.1 (250.5)	4231.5 (159.9)
X25519MLKEM768	ECDSA-SecP256r1-SHA256	522.4 (71.5)	560.1 (69.3)	571.6 (44.2)	684.5 (50.8)	1791.9 (102.3)	871.4 (72.1)
X25519MLKEM768	Ed25519	450.9 (59.7)	488.9 (61.5)	509.1 (42.7)	615.3 (48.3)	1676.3 (105.4)	816.3 (69.5)
X25519MLKEM768	ML-DSA-44	554.9 (82.6)	621.4 (90.4)	659.9 (73.9)	804.5 (75.1)	2131.9 (135.5)	1125.9 (100.2)
X25519MLKEM768	ML-DSA-65	697.8 (114.3)	769.0 (114.5)	793.8 (93.8)	992.9 (101.6)	2386.8 (161.9)	1296.6 (130.0)
X25519MLKEM768	ML-DSA-87	829.3 (122.7)	915.5 (139.2)	963.3 (101.2)	1124.3 (106.9)	2802.8 (182.3)	1531.7 (139.0)
ML-KEM-512	ML-DSA-44	497.9 (80.7)	561.5 (86.6)	594.2 (71.4)	707.7 (73.2)	2123.8 (144.1)	1032.5 (88.2)
ML-KEM-512	ML-DSA-65	636.8 (103.9)	700.7 (106.6)	749.5 (93.6)	876.8 (94.5)	2432.7 (177.6)	1231.3 (111.4)
ML-KEM-512	ML-DSA-87	757.3 (109.3)	840.4 (123.6)	914.8 (108.4)	1023.4 (107.3)	2860.8 (197.3)	1459.4 (119.8)
ML-KEM-768	ML-DSA-44	513.3 (78.4)	580.6 (88.6)	616.3 (69.9)	728.4 (76.2)	2035.6 (135.1)	1073.7 (95.3)
ML-KEM-768	ML-DSA-65	659.9 (111.7)	722.9 (108.8)	749.2 (85.2)	909.6 (94.5)	2290.4 (158.3)	1245.1 (117.2)
ML-KEM-768	ML-DSA-87	779.0 (112.1)	846.4 (122.1)	934.7 (108.2)	1050.6 (107.5)	2722.5 (167.0)	1497.1 (139.1)
ML-KEM-1024	ML-DSA-44	533.0 (78.6)	592.7 (88.2)	686.4 (77.5)	757.8 (81.4)	2100.2 (142.6)	1103.8 (101.8)
ML-KEM-1024	ML-DSA-65	677.5 (112.3)	731.6 (108.6)	846.2 (99.2)	934.7 (96.3)	2345.4 (163.6)	1266.8 (121.5)
ML-KEM-1024	ML-DSA-87	816.3 (111.2)	882.0 (135.5)	996.4 (107.0)	1072.5 (103.9)	2752.5 (161.5)	1497.4 (128.1)

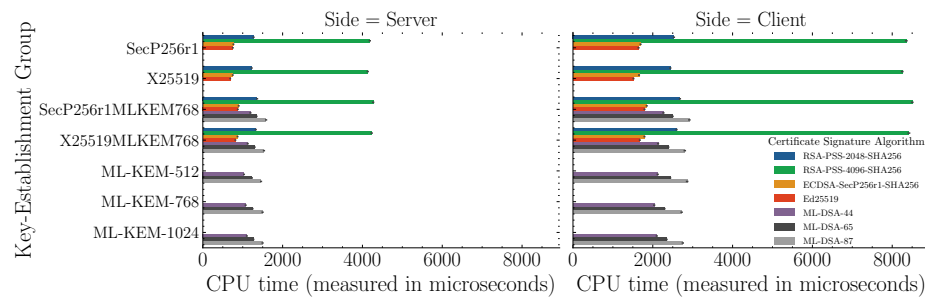


Fig. 9: CPU Time required for mQUIC handshake on client-side and server-side with different key-establishment groups and certificate signature algorithms