**Connessioni TLS post-quantum**

# INTRODUZIONE

L’implementazione degli algoritmi post-quantum si basa sul progetto open source **Open Quantum Safe (OQS)**, scelto per la sua documentazione dettagliata e la disponibilità di strumenti pratici. OQS mette a disposizione la libreria in C denominata **liboqs**, che consente lo sviluppo a basso livello degli algoritmi post-quantum. Grazie a questa libreria, è possibile eseguire test di benchmarking sugli algoritmi di firma digitale e di cifratura post-quantum.

Il progetto include anche il repository **oqsprovider**, un provider per OpenSSL (versione 3.0 e successive) che integra la libreria **liboqs** all’interno di OpenSSL. Questa integrazione semplifica l’adozione degli algoritmi post-quantum, rendendo più agevole l’utilizzo della crittografia quantum-safe in applicazioni reali.

Attualmente, il provider **oqsprovider** supporta la crittografia post-quantum per la negoziazione delle chiavi utilizzando schemi **KEM** in **TLS 1.3**, includendo sia schemi ibridi (post-quantum combinati con algoritmi tradizionali) sia schemi esclusivamente post-quantum. Inoltre, supporta algoritmi di firma ibridi e post-quantum, utilizzabili in contesti come **CMS** (Cryptographic Message Syntax) e **CMP** (Certificate Management Protocol), grazie all’integrazione con l’interfaccia EVP di OpenSSL.

La gestione e la persistenza delle chiavi sono garantite tramite il meccanismo di encode/decode, le strutture dati **X.509** e i pacchetti **PKCS#12**, che consentono di associare una chiave privata al relativo certificato X.509. Con l’introduzione di OpenSSL 3.2, è stato aggiunto il supporto per l’utilizzo delle firme post-quantum in **TLS 1.3**.

# STRUTTURA DEL PROGETTO

OQS fornisce inoltre immagini Docker nel suo repository su Docker Hub, facilitando l’integrazione della crittografia post-quantum in progetti più ampi. Per questo motivo, lo sviluppo delle connessioni **TLS 1.3 post-quantum** seguirà un approccio completamente dockerizzato. In particolare, il progetto utilizzerà i seguenti container:

* Un container contenente un server Flask, scelto per semplicità;
* Un container dedicato alla generazione dei certificati necessari;
* Un container con **Nginx** configurato per gestire le connessioni **TLS 1.3 post-quantum**, includendo il supporto agli algoritmi post-quantum integrati con oqsprovider;
* Un container che permette di interrogare il server tramite **Curl** per verificare il corretto funzionamento delle connessioni TLS;
* Un container con Wireshark, configurato per analizzare dettagliatamente il traffico TLS 1.3 post-quantum e decifrare i messaggi.

Container Flask: Simple HTTP Server

Questo container ospita un server Flask minimale, progettato per rispondere a richieste HTTP con un messaggio JSON, ed è utilizzato per testare connessioni HTTPS in combinazione con i certificati post-quantum generati nel progetto. L’immagine base utilizzata è **Python 3.9-slim**, scelta per la sua leggerezza e per includere tutto il necessario per eseguire un’applicazione Flask. La directory di lavoro è impostata su /app, dove vengono copiati i file dell’applicazione, inclusi il codice del server e il file requirements.txt per l’installazione delle dipendenze.

Il server Flask, implementato nel file app.py, è configurato per avviarsi automaticamente con il comando python app.py e ascolta sulla porta 5000. Il codice definisce un endpoint di base (/) che risponde con un messaggio JSON: {"message": "Hello, HTTPS world!"}.

Il container rappresenta il server applicativo di base per testare e dimostrare la funzionalità delle connessioni **TLS 1.3 post-quantum**, integrate nel progetto tramite il container Nginx.

Generazione dei certificati

Per la generazione dei certificati è stato creato un container denominato **“cert-generator”**, basato su un’immagine Docker ufficiale rilasciata da OQS. Questa immagine include una distribuzione Linux con **OQS-Provider** preconfigurato e funzionante, semplificando l’utilizzo degli strumenti per la crittografia post-quantum.

La generazione dei certificati richiede un file di configurazione, chiamato **“openssl.cnf”**, che viene passato al container tramite un volume condiviso. A seconda delle esigenze, è possibile generare un certificato semplice per il server o, in alternativa, un certificato server firmato da una CA fittizia, ideale per contesti in cui è richiesto un livello di fiducia maggiore.

Per distinguere il certificato del server da quello della CA, nel file di configurazione **openssl.cnf** sono state aggiunte sezioni specifiche: **“v3\_req”** per il certificato del server e **“v3\_ca”** per il certificato della CA. Queste sezioni contengono le impostazioni necessarie per definire le caratteristiche di ciascun certificato. Durante il processo di generazione, l’opzione **-extensions** consente di richiamare la configurazione appropriata, garantendo che il certificato sia conforme al ruolo specifico per cui è stato creato.

La scelta dell’algoritmo di firma post-quantum è ricaduta su **Dilithium5**, sebbene sia possibile utilizzare qualsiasi altro algoritmo di firma supportato dalla libreria OQS.

Nel caso in cui si desideri generare un certificato server non firmato da una CA, è sufficiente eseguire il seguente comando:

openssl req -x509 -new -newkey dilithium5 -keyout /certs/server.key -out /certs/server.crt -nodes -days 365 -config certs/oqs\_openssl.cnf -subj "/CN=nginx\_pq" -extensions v3\_req

Se, invece, si opta per un certificato server firmato da una CA fittizia, il processo richiede più passaggi. Per prima cosa, è necessario generare il certificato della CA utilizzando il comando:

openssl req -x509 -new -newkey dilithium5 -keyout /certs/CA.key -out /certs/CA.crt -nodes -days 365 -config certs/oqs\_openssl.cnf -subj "/CN=oqstest CA" -extensions v3\_ca

Successivamente, si procede con la creazione di una richiesta di firma per il certificato del server (CSR):

openssl req -new -newkey dilithium5 -keyout /certs/server.key -out /certs/server.csr -nodes -config certs/oqs\_openssl.cnf -subj "/CN=nginx\_pq" -extensions v3\_req

Infine, si firma la richiesta CSR con il certificato e la chiave privata della CA per ottenere il certificato del server firmato:

openssl x509 -req -in /certs/server.csr -out /certs/server.crt -CA /certs/CA.crt -CAkey /certs/CA.key -CAcreateserial -days 365

Infine, i certificati della CA e del server vengono concatenati in un unico file, che sarà utilizzato dal server. Questa operazione può essere eseguita con i seguenti comandi:

cat /certs/server.crt > /certs/qsc-ca-chain.crt

cat /certs/CA.crt >> /certs/qsc-ca-chain.crt

Il **Common Name (CN)** nei certificati ha un ruolo importante e richiede attenzione nella configurazione. Per il certificato della **CA**, il CN può essere definito liberamente, poiché non è coinvolto nella verifica diretta del collegamento. Al contrario, per il certificato del **server**, il CN deve corrispondere obbligatoriamente al nome o al link utilizzato dal client per connettersi, ad esempio tramite l’utility **curl** offerta da Linux.

In un progetto Docker, i container comunicano utilizzando il nome del container come hostname. Di conseguenza, se il nome del container cambia, anche il CN del certificato del server deve essere aggiornato per garantire il corretto funzionamento delle connessioni e il superamento della verifica da parte del client.

Per automatizzare l’intero processo, queste istruzioni sono state incluse in uno script denominato **generate\_certs.sh**, che viene passato al container **cert-generator** tramite un volume Docker. Lo script non solo esegue la generazione dei certificati, ma verifica anche la loro esistenza, evitando la sovrascrittura in caso di esecuzioni successive.

Inoltre, configurando opportunamente una variabile d’ambiente all’interno del file **.env**, è possibile ottenere informazioni dettagliate sui certificati appena generati nei log del container, rendendo il processo più trasparente e facilmente monitorabile

Container Nginx: Proxy inverso per connessioni TLS 1.3 post-quantum

Il container che ospita **Nginx** è creato utilizzando un’immagine di Nginx opportunamente modificata per supportare algoritmi di crittografia post-quantum. Questa immagine, rilasciata dal progetto **Open Quantum Safe (OQS)**, integra il supporto per la libreria **oqs-provider**, rendendo possibile l’uso di algoritmi post-quantum come kyber512 e x25519\_mlkem512 per la negoziazione delle chiavi.

La configurazione di Nginx è definita nel file nginx.conf, che specifica i parametri essenziali per il funzionamento del server HTTPS. Il server opera esclusivamente sulla porta **4433** e utilizza il protocollo **TLS 1.3**. Nginx funge da **proxy inverso**, inoltrando le richieste HTTPS al server Flask, accessibile all’indirizzo http://app:5000. Questa configurazione permette di crittografare tutte le comunicazioni tra client e server applicativo, sfruttando la crittografia avanzata post-quantum offerta dal progetto OQS.

I certificati generati dal container **cert-generator** sono fondamentali per il funzionamento di Nginx. Tuttavia, l'ordine di avvio dei container Docker è casuale, il che può causare problemi: se i certificati non sono disponibili al momento dell'avvio di Nginx, ad esempio perché non sono stati generati in una precedente esecuzione o perché **cert-generator** non ha completato la propria attività prima dell'avvio di Nginx, quest'ultimo non riuscirà a partire. Per evitare questa situazione, nel file **docker-compose.yml** è stata introdotta una dipendenza che garantisce che Nginx venga avviato solo dopo che **cert-generator** ha completato con successo la generazione dei certificati.

Container Curl: Verifica delle Connessioni TLS 1.3 Post-Quantum

Il container utilizza un’immagine di Curl, rilasciata dal progetto **Open Quantum Safe (OQS)**, appositamente modificata per supportare algoritmi di crittografia post-quantum. Questa immagine permette di eseguire test avanzati sulle connessioni **TLS 1.3 post-quantum**, sfruttando le funzionalità offerte dalla libreria OQS. L’uso di questa immagine è necessario poiché i browser tradizionali non supportano, al momento, certificati basati su algoritmi di firma post-quantum. Sebbene la cifratura tramite KEM di tipo post-quantum sia già compatibile con alcuni browser, l’assenza di supporto per le firme richiede strumenti dedicati come la versione modificata di Curl fornita da OQS.

Il container è progettato per verificare il corretto funzionamento delle connessioni stabilite dal server Nginx. Dopo essersi assicurato che quest’ultimo sia operativo, utilizza Curl per stabilire connessioni HTTPS sicure, specificando l’algoritmo post-quantum **x25519\_mlkem512** per la negoziazione delle chiavi. Durante i test, il certificato del server viene validato utilizzando il certificato della CA generato dal container **cert-generator**.

L’output della connessione viene registrato in un file di log, mentre un report dettagliato in formato Markdown viene generato per evidenziare le informazioni chiave, come i dettagli dell’handshake TLS e la risposta HTTP ricevuta. Questo container rappresenta uno strumento fondamentale per verificare la configurazione delle connessioni **post-quantum** e monitorare il comportamento del sistema con precisione e trasparenza.

Container Wireshark: Analisi del traffico TLS 1.3 post-quantum

L'immagine utilizzata è stata prelevata dalla repository ufficiale di Docker Hub di Open Quantum Safe (OQS) ed è una versione di Wireshark opportunamente modificata per supportare gli algoritmi post-quantum. Poiché Wireshark dispone di un'interfaccia grafica, l’accesso alla GUI non è diretto, essendo tutto dockerizzato. Per ovviare a questo problema, la visualizzazione grafica viene gestita tramite VcXsrv Windows X Server, che consente il reindirizzamento del traffico X11 al sistema host. La configurazione del container prevede l’utilizzo della modalità di rete "host" per permettere a Wireshark di catturare il traffico direttamente dall’host e include volumi condivisi per salvare i file di cattura nella directory /pcap.

Per evitare di catturare un numero eccessivo di pacchetti e generare rumore inutile, Wireshark è stato configurato per monitorare esclusivamente il traffico diretto a una specifica porta TCP (4433) e proveniente da un particolare indirizzo IP (192.168.1.100). Dato che Docker assegna dinamicamente gli indirizzi IP ai container, è stato necessario configurare un indirizzo IP statico per il container Nginx, in modo da garantire che Wireshark possa monitorare correttamente il traffico senza problemi legati a variazioni nell’indirizzo IP del server.

Il payload delle comunicazioni TLS risulta cifrato, rendendo impossibile ispezionare nel dettaglio lo scambio di messaggi del protocollo TLS senza ulteriori configurazioni. Per consentire l’analisi approfondita, è stato aggiunto un volume al container **Curl**, che permette di registrare i segreti della sessione TLS. Grazie alla variabile d'ambiente **SSLKEYLOGFILE**, il client Curl registra i segreti di sessione TLS nel file /secrets/sslkeys.log. Questo file viene montato nel volume condiviso /secrets, rendendolo disponibile per l’analisi successiva. Una volta generato, il file dei segreti può essere passato a Wireshark per permettere la decifratura completa del traffico TLS e la corretta visualizzazione dei pacchetti scambiati.

# parametri della connessione TLS 1.3 post-quantum

Di seguito viene analizzata una piccola traccia catturata con Wireshark. Il traffico catturato include il messaggio **Client Hello**, ossia il primo scambio del protocollo TLS utilizzato dal client per avviare la negoziazione di una connessione sicura. Questo messaggio contiene tutte le informazioni necessarie per negoziare i parametri di sicurezza, come le suite di cifratura supportate, i gruppi crittografici e le estensioni richieste.

Immagine che contiene testo, schermata, software, numero

Descrizione generata automaticamente

Il messaggio inizia con un **Record Header**, che identifica il tipo di contenuto, la versione del protocollo e la lunghezza dei dati. Il tipo di contenuto è indicato come Handshake (22), che conferma che il record fa parte del processo di handshake. Sebbene la connessione negoziata sarà basata su TLS 1.3, la versione del protocollo riportata è TLS 1.0 (0x0301), una scelta intenzionale per garantire compatibilità con implementazioni precedenti. La lunghezza del record è di 1112 byte, specificando la dimensione del payload che segue.

Il **Handshake Protocol** è identificato come Client Hello (1), un messaggio fondamentale che avvia il processo di negoziazione. La lunghezza totale del messaggio è di 1112 byte. Sebbene il client dichiari la versione TLS 1.2 (0x0303) per motivi di compatibilità, la reale versione negoziata sarà TLS 1.3, comunicata tramite l’estensione "Supported Versions".

Il messaggio include 32 byte di **dati casuali** generati dal client, necessari per il processo di generazione delle chiavi condivise. Inoltre, il client invia un **Session ID** di 32 byte per garantire la compatibilità con dispositivi che potrebbero interpretare la connessione come una sessione TLS 1.2.

Le **Cipher Suites** elencate dal client sono tre: TLS\_AES\_256\_GCM\_SHA384, TLS\_CHACHA20\_POLY1305\_SHA256 e TLS\_AES\_128\_GCM\_SHA256. Questi algoritmi, tutti conformi al design di TLS 1.3, garantiscono crittografia avanzata, autenticità e integrità dei dati. Il client dichiara anche di non utilizzare compressione, specificando il metodo null per evitare vulnerabilità note come CRIME.

Un elemento chiave del Client Hello è rappresentato dalle **estensioni**, che permettono di comunicare funzionalità aggiuntive al server. Tra queste, l’estensione **Server Name Indication (SNI)** specifica il nome dell’host nginx\_pq, essenziale per configurazioni con più host virtuali su un singolo indirizzo IP. L’estensione **Supported Groups** elenca i gruppi crittografici supportati dal client per la negoziazione delle chiavi. L’estensione **Application Layer Protocol Negotiation (ALPN)** indica i protocolli applicativi supportati, come HTTP/1.1. L’estensione **Supported Versions** comunica che il client supporta TLS 1.3, mentre l’estensione **Key Share** include una chiave pubblica generata utilizzando l’algoritmo post-quantum Kyber512, che rappresenta un elemento fondamentale per la sicurezza della negoziazione.

Successivamente, il server risponde con il messaggio **Server Hello**, che conferma i parametri negoziati e avvia la configurazione della connessione sicura.

Immagine che contiene testo, schermata, software, numero

Descrizione generata automaticamente

Anche questo messaggio inizia con un **Record Header**, che identifica il contenuto come parte del processo di handshake (Handshake (22)). Come per il Client Hello, la versione del protocollo indicata è TLS 1.0 (0x0301) per garantire compatibilità con implementazioni precedenti, sebbene la connessione effettiva sarà basata su TLS 1.3. La lunghezza del record è di 858 byte.

Il **Server Hello**, identificato come handshake di tipo Server Hello (2), ha una lunghezza complessiva di 854 byte. Il server dichiara la versione del protocollo come TLS 1.2 (0x0303) per motivi di compatibilità, ma conferma l’utilizzo di TLS 1.3 tramite l’estensione "Supported Versions". Il messaggio include 32 byte di **dati casuali**, 8e87db79dfa57dfb66f7c5e58720250bd42ef4b402eeb400e9ecc27772f74b, necessari per il processo di generazione delle chiavi condivise. Inoltre, il **Session ID** inviato dal client viene restituito inalterato, garantendo la compatibilità con dispositivi che interpretano la connessione come una sessione TLS 1.2.

Il server seleziona la suite di cifratura TLS\_AES\_256\_GCM\_SHA384 (0x1302) tra quelle proposte dal client. Questa suite di cifratura assicura un alto livello di sicurezza, garantendo crittografia avanzata e integrità dei dati. Anche il server conferma che non verrà applicata compressione, scegliendo il metodo null (0) per evitare vulnerabilità legate alla compressione.

Il **Server Hello** include inoltre diverse estensioni. Tramite l’estensione "Supported Versions", il server conferma l’utilizzo di TLS 1.3 per la connessione. L’estensione "Key Share" include una chiave pubblica generata utilizzando Kyber512, un algoritmo di crittografia post-quantum. Questa chiave, lunga 772 byte, consente di completare la negoziazione delle chiavi condivise e garantisce la resilienza agli attacchi crittografici futuri.

Il server invia successivamente un messaggio **Change Cipher Spec**, che indica che i messaggi successivi saranno cifrati utilizzando i parametri appena negoziati. Sebbene non abbia un impatto diretto su TLS 1.3, questo messaggio è incluso per motivi di compatibilità con implementazioni esistenti.

Infine, il server invia il messaggio **Encrypted Extensions**, che completa la configurazione della connessione. Questo messaggio include informazioni relative alle estensioni configurate, come la conferma del nome del server tramite l’estensione "Server Name Indication (SNI)" e il protocollo applicativo negoziato (HTTP/1.1) tramite l’estensione "Application Layer Protocol Negotiation (ALPN)".

Successivamente al messaggio **Server Hello**, il server invia il messaggio **Certificate**, che rappresenta una fase cruciale del processo di handshake TLS. Questo messaggio contiene il certificato digitale utilizzato dal server per autenticarsi nei confronti del client, fornendo tutte le informazioni necessarie per la verifica della sua identità.

Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente

Il certificato principale inviato dal server è identificato nel campo **Subject**, con il valore del **commonName (CN)** pari a nginx\_pq. Questo dettaglio conferma che il certificato è intestato al server richiesto, come specificato dal client nel messaggio **Client Hello** tramite l’estensione Server Name Indication (SNI). Il certificato include una chiave pubblica necessaria per la negoziazione delle chiavi condivise, riportata nel campo **subjectPublicKeyInfo**, che specifica l’algoritmo utilizzato e il valore effettivo della chiave. Il campo **algorithm** indica 1.3.9999.7.3, che corrisponde a **p384\_mldsa65**, un algoritmo che combina tecniche classiche e post-quantum per garantire robustezza e compatibilità. La chiave pubblica stessa è contenuta nel campo **subjectPublicKey** e sarà utilizzata dal client per completare la negoziazione della chiave condivisa.

Il certificato è firmato digitalmente, e l’algoritmo utilizzato per questa firma è riportato nel campo **signatureAlgorithm**, anch'esso con valore 1.3.9999.7.3, corrispondente a **p384\_mldsa65**. Questa scelta garantisce l’integrità e l’autenticità del certificato, proteggendolo da modifiche non autorizzate e assicurando che esso provenga dall’autorità emittente dichiarata.

L’autorità emittente è identificata nel campo **issuer**, dove il valore del **commonName (CN)** è oqstest CA. Questo dettaglio conferma che il certificato è stato emesso da una CA utilizzata per testare connessioni TLS post-quantum. Il certificato, inoltre, specifica un periodo di validità nel campo **validity**, che include le date di inizio e di scadenza, stabilendo l’intervallo temporale entro il quale è considerato valido.

Dopo l'invio del certificato, il server procede con il messaggio **Certificate Verify**, un elemento essenziale per garantire l'autenticità del certificato precedentemente fornito.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Questo messaggio, classificato come Handshake Protocol: Certificate Verify, ha un tipo di handshake identificato come Certificate Verify (15) e una lunghezza complessiva di 3.421 byte.

Il **Certificate Verify** contiene la firma digitale del server, calcolata utilizzando la chiave privata associata al certificato inviato. La firma è riportata nel campo **Signature** ed è utilizzata per validare l'integrità dei messaggi precedenti e l’autenticità del server stesso. Tuttavia, dall’immagine si nota che il campo **Signature Algorithm** appare come Unknown Unknown (0xfed5). Questo comportamento non è dovuto a un errore nella connessione, ma alla scelta di utilizzare una versione standard di Wireshark non modificata per questa fase di analisi. Poiché gli algoritmi post-quantum non sono riconosciuti di default da Wireshark, il campo relativo all'algoritmo risulta sconosciuto. La dimensione della firma, invece, è chiaramente indicata, pari a 3.417 byte, confermando che il messaggio è stato correttamente generato.

Questo approccio è stato adottato per ottimizzare l’usabilità durante l’analisi. Infatti, sebbene sia stato utilizzato un container Docker con una versione modificata di Wireshark, rilasciata da OQS, per catturare il traffico, questa versione è risultata meno agevole da usare. La necessità di accedere all’interfaccia tramite **X11** e il rallentamento generale del sistema hanno reso questa configurazione poco pratica per lo studio dettagliato. Per questo motivo, la cattura del traffico è stata effettuata con Wireshark modificato, mentre l’analisi dei pacchetti è stata eseguita successivamente su una versione standard di Wireshark installata direttamente sull’host.

Subito dopo il messaggio **Certificate Verify**, il server invia il messaggio **Finished**, che rappresenta la conclusione della fase di handshake. Questo messaggio, identificato come Handshake Protocol: Finished, ha una lunghezza di 69 byte e contiene nel campo **Verify Data** 48 byte di dati crittografati. Il ruolo del **Finished** è quello di dimostrare che tutte le chiavi e i parametri negoziati sono stati concordati in modo coerente tra client e server. Il server utilizza la chiave derivata durante l’handshake per calcolare un hash sui messaggi precedenti e invia il risultato al client, che esegue lo stesso calcolo per verificare la corrispondenza. Questa verifica finale garantisce che il handshake sia stato completato correttamente e che la connessione possa proseguire in modo sicuro.

Dopo il messaggio **Certificate Verify** e il successivo **Finished** inviato dal server, il client invia un nuovo messaggio di tipo **Change Cipher Spec**, seguito dal proprio **Finished**, completando così la fase di handshake.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Questi messaggi rappresentano la conferma finale che entrambe le parti hanno negoziato con successo i parametri della connessione sicura.

Il messaggio **Change Cipher Spec** è un placeholder nel protocollo TLS 1.3, utilizzato principalmente per compatibilità con middleware e versioni precedenti del protocollo. Anche se non ha un ruolo funzionale diretto in TLS 1.3, il suo invio garantisce che dispositivi o software intermediari, progettati per gestire versioni precedenti del protocollo, non blocchino la connessione. Questo messaggio è molto breve, con una lunghezza complessiva di un solo byte, e viene inviato prima di qualsiasi messaggio cifrato.

Successivamente, il client invia il messaggio **Finished**, un elemento critico che conclude il processo di handshake. Questo messaggio, classificato come Handshake Protocol: Finished, contiene un hash crittografico basato su tutti i messaggi precedenti scambiati durante l’handshake. Il campo **Verify Data** include 48 byte di dati, calcolati utilizzando la chiave derivata durante il processo di negoziazione. Quando il server riceve questo messaggio, esegue lo stesso calcolo sul flusso di dati e confronta il risultato con quanto inviato dal client. Se i valori corrispondono, il server conferma che l'handshake è stato completato con successo e che entrambe le parti condividono gli stessi parametri crittografici. L’invio del messaggio **Finished** da parte del client segna la conclusione del processo di handshake TLS. Da questo punto in avanti, la comunicazione tra client e server avviene in modo cifrato utilizzando i parametri negoziati. Questo scambio garantisce non solo la riservatezza e l’integrità dei dati, ma anche l’autenticità delle parti coinvolte.

Dopo il completamento dell’handshake, il server invia un messaggio di tipo **New Session Ticket**, che ha un ruolo fondamentale nell’ottimizzazione delle connessioni future tra le stesse parti.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Questo messaggio consente di stabilire una ripresa della sessione TLS senza dover ripetere l’intero processo di handshake, riducendo significativamente la latenza e il carico computazionale nelle connessioni successive.

Il messaggio contiene diverse informazioni chiave per gestire la sessione. Tra queste, il **Session Ticket Lifetime Hint** indica che il ticket sarà valido per un massimo di 300 secondi, pari a cinque minuti, fornendo al client un intervallo temporale entro cui può riutilizzare il ticket per ristabilire la connessione. Per garantire sicurezza contro attacchi di riproduzione, viene incluso un valore casuale chiamato **Session Ticket Age Add**, che viene combinato con l’età del ticket durante il processo di ripresa della sessione. Inoltre, il messaggio include un **Session Ticket Nonce**, con una lunghezza di 8 byte, che garantisce l’unicità del ticket, prevenendo eventuali riutilizzi non autorizzati.

Il **Session Ticket** stesso ha una dimensione di 240 byte ed è rappresentato come una sequenza di dati crittografati. Questo ticket contiene tutte le informazioni necessarie per riprendere la connessione senza richiedere al server di mantenere uno stato persistente. Infine, il campo relativo alle estensioni nel messaggio è vuoto, indicando che non sono stati inclusi ulteriori dati opzionali.

L’invio del messaggio **New Session Ticket** non ha un impatto immediato sulla connessione corrente, ma prepara il sistema per eventuali connessioni successive tra il client e il server, rendendo il protocollo più efficiente e reattivo. Questa funzionalità è particolarmente significativa in TLS 1.3, dove il ticket crittografato protegge le informazioni della sessione, assicurando che siano utilizzabili solo dal client autorizzato.

L’invio dei messaggi **New Session Ticket** continua con l’aggiunta di un secondo ticket, osservabile nel frame successivo.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Questa situazione è comune in TLS 1.3, dove il server può inviare più session ticket per garantire una maggiore flessibilità e sicurezza nella gestione delle sessioni riprese. Ogni ticket rappresenta una possibilità per il client di ristabilire la connessione senza dover eseguire nuovamente il processo completo di handshake, utilizzando invece i dati crittografati e le chiavi generate durante la sessione iniziale.

Il secondo **New Session Ticket** segue lo stesso formato del precedente, includendo un **Session Ticket Lifetime Hint**, che definisce un periodo di validità massimo per il ticket, e un **Session Ticket Age Add**, che fornisce un valore casuale per prevenire attacchi di riproduzione. Anche in questo caso, il ticket contiene una sequenza di byte crittografati che rappresentano le informazioni della sessione. Questo secondo ticket aumenta ulteriormente la flessibilità del client, permettendogli di selezionare il ticket più appropriato per una futura connessione, in base alle sue necessità e ai parametri di sicurezza.

l messaggio finale catturato nella connessione TLS è un messaggio di tipo **Alert**, inviato per notificare la chiusura della sessione.

Immagine che contiene testo, schermata, linea

Descrizione generata automaticamente

Questo messaggio rappresenta un passaggio standard nel protocollo TLS, utilizzato per segnalare che una delle due parti intende terminare la comunicazione in modo corretto e ordinato.

Il messaggio è classificato come Alert Protocol e specifica un **Level** pari a Warning (1) e una **Description** di Close Notify (0). La combinazione di questi valori indica che il server (o il client) sta avvisando l’altra parte della chiusura imminente della connessione. La lunghezza totale del messaggio è di 19 byte.

La presenza del **Close Notify** è essenziale per garantire che non ci siano fraintendimenti nella comunicazione. Senza questo messaggio, la chiusura della connessione potrebbe essere interpretata come un'interruzione imprevista, che potrebbe compromettere la sicurezza o l’integrità dei dati. In questo caso, l’invio del messaggio segnala che tutte le informazioni necessarie sono state scambiate con successo e che non ci sono ulteriori dati da trasmettere.

Questo ultimo messaggio completa il ciclo della comunicazione TLS, confermando che la connessione è stata chiusa in modo corretto e sicuro. L’uso del **Close Notify**, come definito nelle specifiche TLS, assicura che la sessione venga terminata rispettando i principi di sicurezza e integrità, garantendo una chiusura coordinata e senza ambiguità.

# Analisi delle Prestazioni delle Connessioni TLS 1.3 Post-Quantum

Il primo passo per condurre un'analisi delle prestazioni e un confronto significativo tra connessioni TLS 1.3 post-quantum e pre-quantum è definire chiaramente le metriche di interesse e il metodo per valutarle. Per raggiungere questo obiettivo, è stato necessario personalizzare sia il container contenente Curl, utilizzato come client per analizzare le connessioni, sia il container che ospita Nginx, responsabile della terminazione TLS. Questi container sono stati inglobati in immagini Docker personalizzate per aggiungere le librerie necessarie al corretto funzionamento del codice di analisi, garantendo la flessibilità indispensabile per raccogliere dati accurati.

Analisi lato Client

Per simulare un contesto realistico, è stato deciso che le richieste HTTPS dovessero essere effettuate in parallelo, non sequenzialmente. Vengono eseguite in parallelo attraverso il seguente blocco di istruzioni:  
Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Questo blocco di codice gestisce l’esecuzione parallela di un numero definito di richieste HTTPS, sfruttando la libreria concurrent.futures per creare e controllare un pool di thread. L’uso di ThreadPoolExecutor permette di generare fino a NUM\_REQUESTS thread, ognuno dei quali esegue la funzione execute\_request.

Ogni thread viene creato attraverso executor.submit(execute\_request, i + 1), che invoca la funzione con un identificativo numerico della richiesta. Questa funzione avvia una chiamata HTTPS utilizzando curl, raccogliendo le metriche di connessione.

Le richieste vengono gestite in modo asincrono: il ciclo as\_completed(futures) monitora l’avanzamento dei thread e raccoglie i risultati non appena ogni richiesta viene completata, senza aspettare la conclusione delle altre.

Un aspetto importante da considerare è che, sebbene il codice utilizzi un ciclo for per avviare le richieste, queste non vengono eseguite in modo strettamente sequenziale. Infatti, vengono avviate in rapida successione, ma è il ThreadPoolExecutor a gestirne l'esecuzione concorrente, distribuendole tra i thread disponibili. In particolare, il ciclo scorre rapidamente su range(NUM\_REQUESTS) e, a ogni iterazione, executor.submit() avvia un nuovo thread senza attendere la terminazione di quello precedente, creando così un effetto di parallelismo parziale.

A questo punto si procede con un’analisi a due livelli: uno dedicato allo stato generale del sistema e uno focalizzato sulla singola richiesta HTTPS.

Per quanto riguarda il client, le metriche analizzate si concentrano sul tempo necessario per stabilire e completare una connessione HTTPS. Le componenti valutate includono:

* il tempo richiesto per risolvere il DNS e stabilire la connessione TCP (Connect Time);
* il tempo necessario per completare la negoziazione TLS, inclusi lo scambio di chiavi e la verifica dei certificati (TLS Handshake);
* il tempo complessivo della connessione, comprensivo di trasferimento dei dati HTTP (Total Time);
* il tempo totale percepito dal client, misurato dall’invio della richiesta fino al ricevimento della risposta completa (Elapsed Time).

CURL\_COMMAND\_TEMPLATE = ["curl", "--tlsv1.3", "--curves", "x25519\_mlkem512", "--cacert", "/opt/certs/CA.crt", "-w", "Connect Time: %{time\_connect}, TLS Handshake: %{time\_appconnect}, Total Time: %{time\_total}, %{http\_code}\n","-s", "https://nginx\_pq:4433"]

I primi tre tempi sono facilmente ricavabili utilizzando le opzioni offerte da Curl nel seguente modo:

L’ultimo (l’elapsed time) invece è stato misurato avviando un timer all’inizio della richiesta HTTPS e fermandolo al momento della ricezione della risposta, ottenendo così il valore desiderato. Successivamente questi parametri vengono elaborati per poter permettere la corretta visualizzazione nel log del container e nel file “request\_client.csv”.

Per garantire un'analisi accurata delle prestazioni, il sistema deve operare in condizioni **failure-free**, ovvero prive di guasti. Questo perché la presenza di errori comporterebbe la necessità di analizzare la **performability**, che combina aspetti di prestazioni e affidabilità, valutando non solo le prestazioni del sistema, ma anche la sua capacità di mantenere un livello accettabile di servizio nonostante eventuali guasti. Per assicurarsi che le misurazioni considerino solo scenari **failure-free**, ogni richiesta viene monitorata attraverso il codice di risposta HTTP: se il codice restituito è **200**, significa che la richiesta è stata completata con successo. In questo modo, eventuali fallimenti vengono rilevati e scartati dall'analisi, evitando di compromettere la valutazione delle prestazioni pure. Per facilitare il monitoraggio delle richieste HTTPS andate a buon fine, è stato inoltre introdotto un contatore che ne tiene traccia.

Un’altra metrica analizzata riguarda il volume di dati effettivamente scambiati in **upload** e **download** tra il client e Nginx. Per misurare questa metrica, è stato utilizzato lo strumento **"trace"** di cURL, che consente di registrare, per ogni singola richiesta, un log dettagliato contenente i messaggi scambiati e la loro dimensione.

Dopo l’esecuzione del test, questi log possono essere consultati all’interno del container del client, nel percorso **/app/logs/**. In particolare, all’interno del log, risultano di interesse cinque righe specifiche: **due riguardano i dati inviati dal client al server**, mentre **tre fanno riferimento ai dati ricevuti dal client dal server**.

Immagine che contiene testo, ricevuta, Carattere, bianco

Descrizione generata automaticamente

Immagine che contiene testo, Carattere, ricevuta, bianco

Descrizione generata automaticamente Nel primo caso, vengono registrati i messaggi trasmessi dal client verso il server. Per quantificare il volume totale dei dati scambiati in questa direzione, si esegue una ricerca all’interno del log per identificare queste righe e sommare le dimensioni dei messaggi corrispondenti.

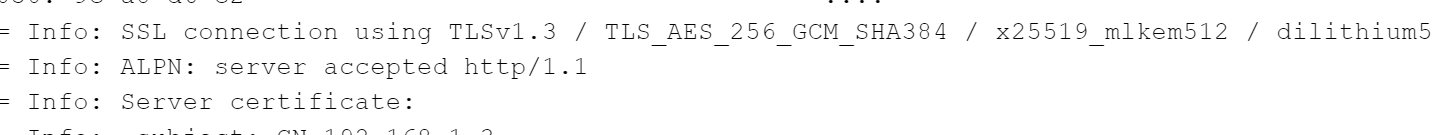
Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente

Nel secondo caso, il procedimento è analogo al primo, con la differenza che in questo scenario vengono considerati i **byte scambiati dal server verso il client**. Anche in questo caso, si effettua una ricerca all'interno del log per individuare le tre righe di interesse e sommare le dimensioni dei messaggi registrati, così da ottenere il totale dei dati trasferiti dal server al client.

Infine, per garantire un'analisi completa e accurata, nel file request\_client.csv vengono registrati anche l'algoritmo di firma, il KEM e la dimensione del certificato utilizzati per stabilire la connessione. La dimensione del certificato viene determinata nello stesso modo in cui viene calcolata la quantità di byte scambiati, sfruttando una specifica riga presente nel file di log di trace:  
Immagine che contiene testo, ricevuta, Carattere, bianco

Descrizione generata automaticamente

Sebbene il KEM e l’algoritmo di firma siano già noti, in quanto parametri necessari per effettuare la richiesta, è comunque utile verificarne la correttezza. Per questo motivo, viene effettuata una ricerca all’interno del log di trace per individuare una specifica riga che riporta gli algoritmi effettivamente utilizzati durante la connessione:

Per quanto riguarda le metriche globali del client, l’analisi si concentra sul monitoraggio delle risorse di sistema, con un’attesa di circa **0,1 secondi** tra una rilevazione e l’altra. Tuttavia, poiché il tempo di attesa dipende anche dall’esecuzione del codice stesso, il campionamento effettivo **non ha una precisione assoluta di 0,1 secondi**, ma risulta comunque sufficientemente frequente da fornire una rappresentazione accurata dell’andamento delle risorse. Il processo di monitoraggio termina automaticamente quando, per cinque rilevazioni consecutive, non vengono rilevate connessioni attive. I dati raccolti vengono salvati in un secondo file CSV denominato *system\_client.csv*.

Le metriche monitorate includono l’utilizzo della CPU, della memoria e il numero di connessioni TLS attive in quel momento. Queste informazioni vengono ottenute tramite la libreria **psutil**, disponibile nell’ambiente Python, che consente di misurare in tempo reale il consumo delle risorse di sistema.

Le rilevazioni effettuate riguardano esclusivamente le risorse impiegate per le richieste HTTPS e non tengono conto di altri processi attivi che, nello stesso momento, utilizzano le risorse dell’host. Ciò avviene perché l’ambiente Docker fornisce un isolamento rispetto al resto del sistema, e le misurazioni si riferiscono unicamente all'ambiente Docker.

Analisi lato Nginx

Per quanto riguarda Nginx, non è stato possibile effettuare un’analisi delle metriche a grana fine, ovvero valutando l’impatto di ogni singola richiesta. Questa limitazione è dovuta al fatto che Nginx si occupa di ricevere e gestire le richieste, ma non le genera autonomamente.

Per condurre un’analisi dettagliata a livello di singola richiesta, sarebbe necessario un processo sempre attivo che monitori continuamente il traffico in ingresso, avviando per ogni richiesta un’istanza del codice dedicata alla raccolta delle metriche. Tuttavia, questa soluzione risulterebbe poco praticabile, in quanto introdurrebbe un impatto significativo sull’utilizzo delle risorse di Nginx, compromettendo l'affidabilità dei dati raccolti.

Inoltre, monitorare una singola richiesta potrebbe portare a risultati distorti a causa della natura parallela delle connessioni. Durante il campionamento delle risorse relative a una specifica richiesta, altre richieste potrebbero arrivare simultaneamente, alterando i valori rilevati e rendendo difficile isolare l’effettivo impatto di una singola connessione. Per queste ragioni, si è deciso di concentrarsi esclusivamente sulle metriche globali.

Le metriche globali monitorate includono l’utilizzo della CPU e della memoria, il numero di connessioni TLS attive in quel momento e il quantitativo di byte scambiati. Queste informazioni vengono ottenute tramite la libreria *psutil* (la stessa utilizzata nel client), disponibile nell’ambiente Python, che consente di misurare in tempo reale il consumo delle risorse di sistema.

Come per il client, anche per Nginx le rilevazioni effettuate riguardano esclusivamente le risorse impiegate per le richieste HTTPS e non tengono conto di altri processi attivi che, nello stesso momento, utilizzano le risorse dell’host. Ciò avviene perché l’ambiente Docker fornisce isolamento rispetto al resto del sistema, e le misurazioni si riferiscono unicamente all’ambiente Docker.

È importante sottolineare che il quantitativo di byte scambiati corrisponde al totale dei dati inviati e ricevuti dall’avvio del sistema fino al momento della rilevazione. Di conseguenza, questa metrica include non solo i dati relativi alle richieste HTTPS, ma anche il traffico generato dall’installazione delle librerie necessarie. Tuttavia, l’influenza di questo traffico iniziale è facilmente individuabile nel file *monitor\_nginx.csv*, poiché le richieste HTTPS vengono avviate solo dopo che Nginx ha avviato il proprio script. Di conseguenza, i byte scambiati nei primi campionamenti sono attribuibili esclusivamente alle operazioni di preparazione dell’ambiente e non all’effettiva esecuzione delle richieste HTTPS.

A questo punto, sorge la necessità di correlare il monitoraggio delle risorse con le richieste HTTPS in arrivo. Per ottenere questa associazione, è stata adottata la seguente strategia: il monitoraggio viene eseguito **indipendentemente dalle richieste**, con un’attesa di circa **0,1 secondi** tra una rilevazione e l’altra. Tuttavia, come per il client, poiché il tempo di attesa dipende anche dall’esecuzione del codice stesso, il campionamento effettivo **non ha una precisione di 0,1 secondi esatti**, ma risulta comunque sufficientemente frequente da fornire una rappresentazione accurata dell’andamento delle risorse.

Successivamente, per legare queste misurazioni alle richieste HTTPS, si sfrutta il log generato da **Nginx** al momento della gestione di ogni richiesta. In particolare, il monitoraggio continua fino a quando il log di Nginx non contiene un numero di entry pari al numero di richieste previste (dato che ogni richiesta genera una nuova entry nel log). Una volta raggiunto questo stato, il file di log viene analizzato per individuare il **timestamp minimo e massimo** tra le entry registrate, ottenendo così l’intervallo temporale esatto in cui le richieste sono state servite. Questo range viene quindi applicato alle rilevazioni già effettuate nel file monitor\_nginx.csv, permettendo di filtrare solo le misurazioni di interesse e di salvarle in un nuovo file denominato monitor\_nginx\_filtered.csv.

Tuttavia, questo filtraggio **non è completamente preciso** a causa della risoluzione temporale del log di **Nginx**, che registra i timestamp con una precisione che **non scende sotto il secondo**. Ciò significa che alcune rilevazioni prive di informazioni utili potrebbero comunque essere considerate rilevanti e incluse nell’analisi, limitando leggermente la precisione del filtraggio.

Il monitoraggio delle risorse, insieme al conteggio delle entry nel log di **Nginx**, viene gestito dalla funzione **monitor\_resources()**, che registra le metriche di sistema e monitora il numero di richieste servite.

La funzione **analyze\_logs()** ha il compito di analizzare il file di log di Nginx per individuare il **timestamp minimo e massimo**, determinando così l’intervallo temporale in cui sono state processate le richieste HTTPS.

Per elaborare i dati raccolti, la funzione **load\_resource\_data()** legge le entry registrate nel file CSV generato da **monitor\_resources()**, permettendo di accedere alle rilevazioni delle risorse di sistema.

Infine, la funzione **analyze\_performance()** integra questi dati richiamando dapprima **analyze\_logs()**, per ottenere i timestamp di riferimento, e successivamente filtrando le rilevazioni nel file monitor\_nginx\_filtered.csv. Attraverso un ciclo che confronta i timestamp delle rilevazioni con il range temporale calcolato, vengono selezionate solo le entry comprese tra il **timestamp minimo e massimo**, che vengono quindi salvate nel file monitor\_nginx\_filtered.csv.