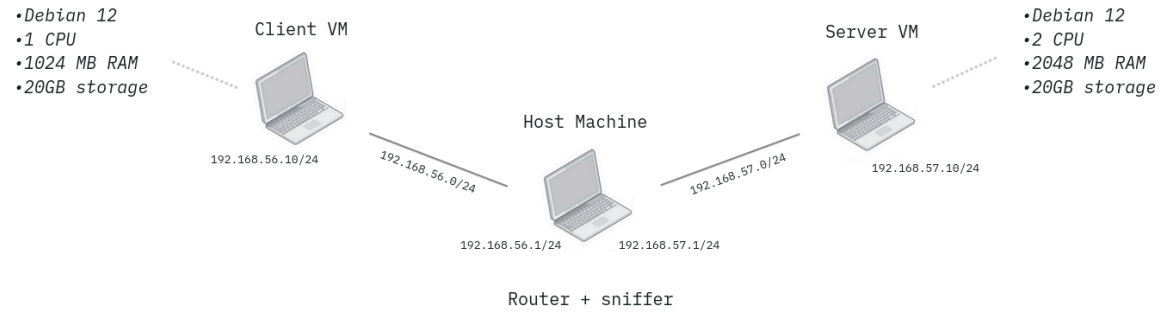
Architecture

The architecture used for this experiment is the one shown in the image below.



Each Virtual Machine has been configured with OpenSSL (v3.5.4) paired with liboqs (v0.15.0) to offer PQC support, together with oqsprovider (v0.11.0) to show the liboqs interfaces to OpenSSL.

Setup

The following setup has been carried out in both the Client and the Server Virtual Machines.

1. Install built-in dependencies:

```
sudo apt update
sudo apt install -y \
 build-essential \
 cmake \
 ninja-build \
 git \
 perl \
 pkg-config \
 libssl-dev
```

2. Create the working directory:

```
mkdir -p ~/pqc
cd ~/pqc
```

3. Fetch and extract OpenSSL source code:

```
wget https://www.openssl.org/source/openssl-3.5.4.tar.gz
tar xf openssl-3.5.4.tar.gz
cd openssl-3.5.4
```

4. Configure OpenSSL for a local build and compile it:

```
./Configure linux-x86_64 \
  --prefix=$HOME/pqc/build \
  --openssldir=$HOME/pqc/build/ssl \
  shared
make -j$(nproc)
make install
```

4.5. Update .bashrc in the home directory of the VM. Add at the end:

```
# Load custom PQC OpenSSL (built in ~/pqc)
export LD_LIBRARY_PATH=/home/vm_user_name/pqc/build/lib64:$LD_LIBRARY_PATH
export OPENSSL_CONF="/home/vm_user_name/pqc/build/ssl/openssl.cnf"
export OPENSSL_MODULES="/home/vm_user_name/pqc/build/lib/ossl-modules"
alias openssl_pqc="/home/vm_user_name/pqc/build/bin/openssl"
```

Enable the configuration with:

```
source .bashrc
```

This will load the custom OpenSSL, which we can call with the openssl_pqc custom command.

5. Clone liboqs:

```
cd ~/pqc
git clone https://github.com/open-quantum-safe/liboqs.git
cd liboqs
git checkout 0.15.0
```

6. Configure liboqs as a shared library in our working directory:

```
mkdir build
cd build
cmake .. \
  -DCMAKE_BUILD_TYPE=Release \
  -DBUILD_SHARED_LIBS=ON \
  -DOQS_USE_OPENSSL=OFF \
  -DCMAKE_INSTALL_PREFIX=$HOME/pqc/build
```

7. Compile and install liboqs locally:

```
make -j$(nproc)
make install
```

8. Clone oqs-provider:

```
cd ~/pqc
git clone https://github.com/open-quantum-safe/oqs-provider.git
cd oqs-provider
git checkout 0.11.0
```

9. Configure oqs-provider against the locally built OpenSSL and liboqs and compile:

```
mkdir build
cd build
cmake .. \
  -DOPENSSL_ROOT_DIR=$HOME/pqc/openssl-3.5.4 \
  -Dliboqs_DIR=$HOME/pqc/build/lib/cmake/liboqs \
  -DCMAKE_BUILD_TYPE=Release
make -j$(nproc)
```

10. Copy the oqs-provider library in the custom OpenSSL modules directory:

```
cp ~/pqc/oqs-provider/build/lib/oqsprovider.so ~/pqc/build/lib/
```

11. Create the OpenSSL configuration file with:

```
nano ~/pqc/build/ssl/openssl.cnf
```

And write:

```
openssl_conf = openssl_init

[openssl_init]
providers = provider_section

[provider_section]
default = default_provider
oqsprovider = oqs_provider

[default_provider]
activate = 1

[oqs_provider]
module = /home/vm_user_name/pqc/build/lib/oqsprovider.so
activate = 1
```

12. Create the certificates in the folder ~/pqc/pqc_certificates, and the keys in ~/pqc/pqc_keys with the script gen_certs.sh.

We can test our configuration by running the following commands on the server, assuming the client VM and the server VM use users named client and server:

```
OPENSSL_CONF=/home/server/pqc/build/ssl/openssl.cnf \
openssl_pqc s_server \
-tls1_3 \
-provider default -provider oqsprovider \
-accept 0.0.0.0:4433 \
-cert /home/server/pqc/pqc_certificates/ML-DSA-65.crt \
-key /home/server/pqc/pqc_keys/ML-DSA-65.key \
-groups MLKEM768 \
-no_ticket \
-num_tickets 0 \
-ciphersuites TLS_AES_256_GCM_SHA384 \
-www
```

And on the client (assuming the Server IP is 192.168.57.10) run:

```
OPENSSL_CONF=/home/client/pqc/build/ssl/openssl.cnf \
openssl_pqc s_client \
-tls1_3 \
-provider default -provider oqsprovider \
-connect 192.168.57.10:4433 \
-groups MLKEM768 \
-sigalgs ML-DSA-65 \
-no_ticket \
-ciphersuites TLS_AES_256_GCM_SHA384 \
-keylogfile keys.txt
```

This should perform a TLS handshake with MLKEM768 as a key exchange mechanism, with ML-DSA-65 as a signature algorithm. The AEAD cipher used will be TLS_AES_256_GCM_SHA384, and the negotiated keys will be saved in ./keys.txt.
In case the handshake fails, make sure the Virtual Machines interfaces are configured correctly, and that IP routes ensure that the client and the server can communicate.

Schemes Under Test

The following KEM algorithms will be tested:

- MLKEM512, BIKE-L1, FrodoKEM640-AES, P256 Hybrid MLKEM512, P256 Hybrid BIKE-L1, P256 FrodoKEM640-AES, secp256r1
  [128 bits security level]
- MLKEM768, BIKE-L3, FrodoKEM976-AES, P384 Hybrid MLKEM768, P384 Hybrid BIKE-L3, P384 FrodoKEM976-AES, secp384r1
  [192 bits security level]

The following signature schemes will be tested:

- ML-DSA-44, SPHINCS+-SHA2-128, Falcon padded 512, Mayo 1, P256 Hybrid ML-DSA-44, P256 Hybrid SPHINCS+-SHA2-128, P256 Hybrid Falcon padded 512, P256 Hybrid Mayo 1, ECDSA secp256r1_sha256
  [128 bits security level]
- ML-DSA-65, SPHINCS+-SHA2-192, Mayo 3, P384 Hybrid ML-DSA-65, P384 Hybrid SPHINCS+-SHA2-192, P384 Hybrid Mayo 3, ECDSA secp384r1_sha384
  [192 bits security level]

The round of KEM measurements will involve handshakes negotiating the expressed KEM algorithms paired with a non PQC signature scheme (ECDSA). Standard Diffie Hellman will be used as a baseline.
The round of signature scheme measurements will involve handshakes negotiating the expressed signature scheme algorithms paired with a non PQC KEM (standard Diffie Hellman). Standard ECDSA will be used as a baseline.
The ciphersuite selected will be either TLS_AES_128_GCM_SHA256 or TLS_AES_256_GCM_SHA384, depending on the security level.
The best algorithms will be chosen to complete a final round of full PQC stack vs Hybrid PQC stack vs non PQC stack.

Measurements

To carry out our analysis, we will measure the following quantities:

- total handshake execution time (from TCP Syn message to Client Finished)
- Bytes of payload sent during handshake

- Client CPU and memory usage in each handshake
- Server CPU and memory average usage per handshake

The measurements will be carried out both under zero stress and under a 70% CPU stress load on all available CPUs of the VMs using the *stress-ng* directive.
To prepare the environment for the measurements, place the folder *server_scripts* and *client_scripts* in the *pqc* folder in the virtual machines, and rename them *scripts.*

Server Scripts

1. rosp_server.py

Example usage:

```
python3 rosp_server.py   /
--bind-ip 0.0.0.0   /
--control-port 9000   /
--port 4433   /
--openssl-bin /home/server/pqc/build/bin/openssl   /
--openssl-conf /home/server/pqc/build/ssl/openssl.cnf
```

rosp_server.py is a passive script to be run for all the measurements taken on the client VM.
It will make the server machine listen on the *control-port* input, on which the client is expected to communicate the key exchange mechanism, the digital signature algorithm and the ciphersuite to be supported for the handshakes. This simple protocol will be denoted as Remote OpenSSL Startup Protocol (ROSP), and its usage will be seen over all the client and server scripts.
After receiving a ROSP message, the server will start listening on the specified port (ex. 4433) for TLS handshakes, accepting only to negotiate the specified mechanisms received in the message.

2. server_control_mem_cpu.py

Example usage:

```
python3 /home/server/pqc/scripts/server_control_mem.py \
  --bind-ip 192.168.57.10 \
  --control-port 9000 \
  --port 4433 \
  --openssl-bin /home/server/pqc/build/bin/openssl_pqc \
  --openssl-conf /home/server/pqc/build/ssl/openssl.cnf \
  --sample-interval 0.01
```

server_control_mem_cpu.py extends the passive rosp_server.py script to also measure the average CPU and memory usage per handshake while the server was active.
Note that the server now expects the client to also send in its ROSP message the number of handshakes it will perform (to average the measurements of CPU and memory usage), and a path in which it should save the results of the measurements.

Client Scripts

The client scripts have the job of sending ROSP messages to start the server and to start requesting handshakes. The *client_scripts* folder is divided into puppet and puppeteers scripts.
The puppet scripts launch the handshakes and, if needed, measure data. Each execution of a puppet script launches a specified number of handshakes with a specified KEM, signature scheme, and ciphersuite after sending the corresponding ROSP message. All results are automatically saved in the folder *results*, inside the *pqc* environment.
Each puppeteer executes the corresponding puppet for all of the configurations of KEM, signature scheme and ciphersuites.
To carry out the handshakes it is sufficient to just run all the puppeteers, which have inputs to be provided to the puppets hardcoded in them. The inputs can be adjusted based on the environment.

Example usage of any puppeteer:
python3 puppeteer_name.py

Results

The results can be found in the Results folder of the repository. They are divided by security level, hybridness, and algorithm under test (KEM/signature scheme).
As expected, the best performing algorithms are the already standardized MLKEM for KEM, ML-DSA for signature.
Their timing, CPU and memory performance is comparable (in some cases even a bit better) than the non PQC baseline.

The Sphincs+ family, now standardized to SLH-DSA, shows the worst performance across signature schemes, which is to be expected given the heavy nature of the hash based algorithm. We experience bytes sent and CPU time usage an order of magnitude higher than its peers.

In general, the big difference between PQC and non PQC can be observed in the bytes exchanged, which is in line with the different sizes of the certificates with respect to the non PQC algorithms.

It is noteworthy to observe that hybrid schemes perform consistently worse than pure PQC counterparts, which makes sense given their inherent nature to use two algorithms instead of one to compute keys or signatures.

Conclusions

The measurement results show there are consistent algorithms that offer strong long term protection without paying too much in performance, at least for non constrained devices.
The high memory usage and power needed to send additional bytes might be an important limitation to consider when implementing protection in limited infrastructures, like IoT networks.
The problem is only amplified when considering hybrid alternatives to ensure a smooth transition from a pre-PQC world to a PQC one.
Depending on the security level needed, this analysis shows, nevertheless, that MLKEM paired with ML-DSA (either in a pure or hybrid configuration) appears to be the best choice for a strong, quantum resistant infrastructure.