Scalable and Efficient Hybrid QKD with Post-quantum Authentications

Raymond K. Zhao, Dongxi Liu, and Josef Pieprzyk

CSIRO's Data61
www.csiro.au

Introduction

It is commonly believed that using post-quantum signatures as the QKD authentication method will severely degrade the QKD performance due to its high computational and communication costs. We evaluate the QKD key rate in a QKD simulator with 3 post-quantum signatures standardised by the NIST: Dilithium, Falcon, and SPHINCS+. We show that the key rate with either Dilithium or Falcon is close to the key rate with AES-GCM (PSK) on 128-bit security for the authentication in such scenarios.

Post-quantum Signatures

Performance of the reference C implementations of Dilithium, Falcon, and SPHINCS+ (128-bit security, without AVX2) measured on Intel Xeon E5-2660 v2 at 2.2 GHz is summarised in Table 1.

Scheme	Sign	Verify	Public Key	Signature
Dilithium2	0.84	0.20	1312	2420
Falcon-512	0.54	0.06	897	666
SPHINCS+ -SHAKE-128f	200.1	12.8	32	17088

Table 1: Sign/Verify Speed (ms) at 2.2 GHz and Public Key/Signature Sizes (Bytes) of Post-quantum Signatures.

Methodology

- We use the QKD simulator from Pedone et al. (2021). This simulator has implemented the BB84 protocol by using Qiskit. Network communication is realised by Docker Compose. This simulator has implemented both the AES-GCM authentication (via `cryptography` with OpenSSL backend) and the post-quantum authentication with an old version of SPHINCS+ (via `pyspx`). PSK are stored on a database server, and the public keys of SPHINCS+ are stored on the QKD nodes locally (PKI is not implemented). Error correction and privacy amplification are also not implemented in this simulator.
- We update `pyspx` to the NIST standardised version, and add Dilithium and Falcon signatures to the simulator. Since the simulator is implemented in Python, the post-quantum signatures (including `pyspx`) use their corresponding reference C implementations as backends with Python bindings. Public keys of Dilithium and Falcon are also stored on QKD nodes locally.
- We adapt similar benchmark methodology to Pedone et al. (2021) in order to measure the key rate. We use a simple network with two peer-to-peer QKD nodes and no eavesdroppers. 5 qubits are used in the BB84 protocol, which gives the best key rate in the simulation (Pedone et al., 2021). A QKD manager sets the authentication method and triggers key exchanges between two QKD nodes with different key lengths. For each combination, we measure the average elapsed time of 100 key exchanges on Intel Xeon E5-2660 v2 at 2.2 GHz (without AVX2 and AES-NI) and compute the key rate.

Results

The QKD key rates with different authentication methods and key lengths are shown in Figure 1. In addition, for the same key length, the achieved percentage of key rates with post-quantum authentications compared to AES-GCM is shown in Table 2.

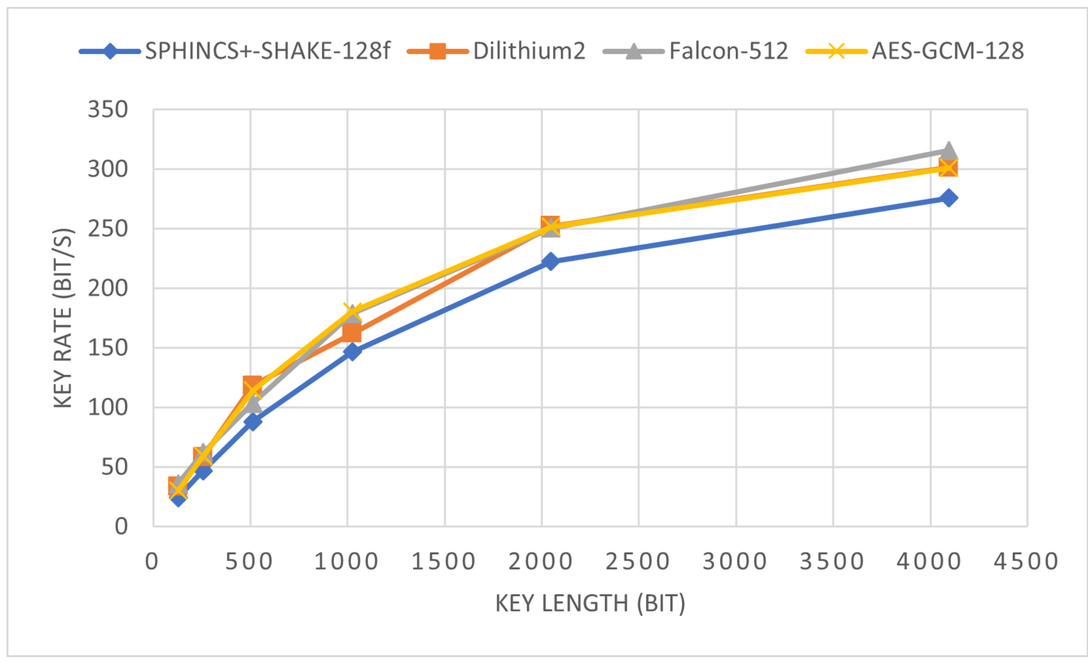

Figure 1: Key rates with different authentication methods.

	128	256	512	1024	2048	4096
Dilithium2	111%	98%	104%	90%	100%	100%
Falcon-512	119%	105%	91%	99%	100%	105%
SPHINCS+ -SHAKE-128f	81%	79%	77%	81%	88%	92%

Table 2: Key Rates Compared to QKD with AES-GCM.

Discussions

- QKD with either Dilithium or Falcon consistently achieves higher key rates compared to SPHINCS+. This is not surprising, given the benchmark results in Table 1.
- The overhead of Dilithium and Falcon compared to AES-GCM is $\leq$2% in most cases ($\leq$10% in all cases), while the overhead of SPHINCS+ is typically $>$10% ($\geq$8% in all cases).
- QKD with Falcon achieves higher key rates than Dilithium in most cases, due to its smaller signature size and faster speed.

FOR FURTHER INFORMATION
Raymond K. Zhao
t: +61 481770413
e: raymond.zhao@data61.csiro.au
w: www.csiro.au

REFERENCES

Pedone et al. (2021). Toward a Complete Software Stack to Integrate Quantum Key Distribution in a Cloud Environment. *IEEE Access*, 9, 115270–115291.

RESOURCES

https://gitlab.com/raykzhao/qkd-sim