Integration of CRYSTALS-Dilithium into the XRP Ledger

Ripple's XRP Ledger is fundamental for securing blockchain transactions efficiently.However, the rise of quantum computing threatens the classical cryptographic algorithms the XRPL relies on, such as secp256k1 and ed25519-donna, for securing transactions and validating digital signatures. Shor's and Grover's algorithms can break the security of these algorithms using quantum computers. Hence, we are proposing the integration of the PQC algorithm: CRYSTALS-Dilithium, alongside existing classical algorithms. This requires modifications to the XRPL protocol, key management infrastructure, core functions etc. and each crucial change that we have made is described in this report. This documentation consists of a short yet technical summary of how we integrated post quantum cryptographic algorithm "CRYSTALS-Dilithium" into the XRP Ledger. The updated code repository with the PQC updates can be found here.

I. Source of Dilithium

We sourced Dilithium from the Dilithium's official repository. Like all other external algorithms, we placed it in rippled/external directory. We have madhe some internal changes to the Dilithium codebase to match the functionality with existing functions within rippled.

In the rippled codebase (rippled/xrpld/protocol/SecretKey.cpp), which consists of all the core functions for cryptographic operations, there exists a function derivePublicKey() which derives a public key from a secret key. This was possible with secp256k1 or ed25519-donna because they all follow Elliptic Curve Cryptography (ECC). However, as Dilithium is a PQC algorithm that relies on lattice-based cryptography, it is not possible to retrieve the public key from the secret key. To address this situation, we came up with an idea to create a new function: *crypto_keypair_seed()* that will generate a key pair deterministically based on the seed provided. So, we just need to store the seed to generate/derive the public key.

The *crypto_sign_keypair* function generates a public and private key pair using internal randomness, making it suitable for scenarios requiring non-deterministic key generation. This was the original function in the repository. In contrast, the *crypto_keypair_seed()* function deterministically derives the same key pair from a given seed, enabling repeatable key generation. This is the new function created. While the former is ideal for generating random key pairs, the latter provides flexibility for scenarios requiring key regeneration from a known seed. For the use case in the XRP Ledger, the second function is very relevant, as we can use it for deterministic as well as non-deterministic key parr generation.

II. Changes to XRP Ledger's codebase

The main changes we have made are primarily in the protocol directory, which includes SecretKey.cpp and PublicKey.cpp files, along with the ancillary changes.

SecretKey.cpp

Destructor: The original implementation securely erased a fixed-size buffer (32 bytes). The modified version supports dynamically allocated buffers (e.g., 32 bytes for Secp256k1, 2528 bytes for Dilithium) using std::vector, ensuring proper memory management and secure erasure.

generateKeyPair(): Support for Dilithium was added alongside Secp256k1 and Ed25519. For Dilithium, the secret key is generated using *generateSecretKey()*, and the public key is derived using an updated *derivePublicKey()* function that accepts a seed.

randomKeyPair(): The original version supported only Ed25519. The modified version explicitly supports Secp256k1, Ed25519, and Dilithium, with tailored logic for each key type to ensure compatibility.

generateSecretKey(): Support for Dilithium was added. The function now uses *crypto_keypair_seed()* for deterministic key generation. Error handling was extended to include std::runtime_error for Dilithium failures, and secure erasure was implemented for all temporary buffers.

derivePublicKey(): The original version supported Secp256k1 and Ed25519. The modified version adds support for Dilithium by introducing an overload that uses *pqcrystals_dilithium2_ref_keypair_seed()* to derive the public key deterministically.

randomSecretKey(): The original function was generic and did not differentiate between key types. The modified version introduces separate functions for Secp256k1, Ed25519, and Dilithium, each tailored to the specific requirements of the respective algorithm.

sign(): Support for Dilithium was added. The function now handles larger signature sizes (e.g., 2420 bytes for Dilithium) and uses *crypto_sign_signature()* for signing. Error handling was extended to ensure compatibility with all supported key types.

SecretKey.h: The buffer representation was changed from a fixed-size array to a dynamically sized std::vector, enabling support for variable key sizes. Constructors and comparison operators were updated to handle dynamic buffers, and hexadecimal conversion was adapted accordingly.

PublicKey.cpp

Key Type Support The original implementation supported only Secp256k1 and Ed25519 for public key construction, verification, and digest verification. The modified version adds support for Dilithium, including:

- Handling Dilithium public keys in the PublicKey constructor with sizes up to CRYPTO_PUBLICKEYBYTES (1312).
- Detecting Dilithium public keys in publicKeyType based on their size.
- Implementing signature verification for Dilithium in *verifyDigest()* and *verify()* using *crypto_sign_verify()*.

PublicKey Constructor The original constructor assumed a fixed public key size of 33 bytes for Secp256k1 and Ed25519. The modified version dynamically determines the public key size based on the key type:

- 33 bytes for Secp256k1 and Ed25519.
- 1312 bytes for Dilithium.

The buffer is resized dynamically to match the key size.

Verification Functions *verifyDigest()*: The original function supported Secp256k1 and Ed25519. The modified version adds Dilithium support using *crypto_sign_verify()* for digest verification.

***verify()*:** The original function verified messages for Secp256k1 and Ed25519. The modified version includes Dilithium message verification using *crypto_sign_verify()*, ensuring compatibility with larger signatures.

Error Handling Error handling was extended to include Dilithium, ensuring proper validation of larger key sizes and signatures in all relevant functions.

Ancillary Changes and Future Work

Build System Updates

- **CMakeLists.txt:** Updated to include build commands for Dilithium, similar to Secp256k1 and Ed25519.
- **Conanfile:** Modified to link libraries required for Dilithium, such as OpenSSL.

Canonicality and Key Parsing Canonicality checks and key parsing for Dilithium are not yet implemented due to the larger key sizes. These features will be addressed in future updates as needed.

A. Modifications to the test cases

To test the functionality of Dilithium, we adhered to a consistent coding standard across all applicable test cases. The test cases have been primarily divided into three types; Class A (least modifications required), Class B (extensive modifications required) and Class C (newly created test case).

III. Test Case Summaries and Performance Analysis

Class A: InnerFormatSerializer_test

Original Implementation: This test validates the serialization and deserialization of Signer objects in multi-signing scenarios using secp256k1. Key steps include:

- Creating a transaction with an account ID and an empty signing public key.
- Generating a second key pair for multi-signing and preparing multi-signing data.
- Testing well-formed Signer objects for successful serialization and malformed objects for rejection.

Modified Implementation: InnerObjectFormatsSerializer_dilithium_test: This test extends support to Dilithium, ensuring robustness in the multi-signing framework. Key changes include:

- Generating Dilithium key pairs for signing transactions.
- Ensuring serialization logic handles larger key and signature sizes.
- Validating well-formed and malformed Signer objects for Dilithium.

Class B: NegativeUNLVoteInternal_test

Original Implementation: This test validates the creation and processing of UNLModify transactions using *secp256k1*. Key steps include:

- Generating *secp256k1* key pairs for validators.
- Verifying that transactions are correctly added to the transaction set.

Modified Implementation: The test now supports Dilithium, ensuring compatibility with larger key sizes. Key changes include:

- Replacing *secp256k1* with *Dilithium* for validator key generation.
- Using *randomSeed()* and *randomDilithiumSecretKey()* for deterministic key pair generation.
- Verifying that Dilithium-based UNLModify transactions are processed seamlessly.

Class C: SecretKeyFunctions_test

This test suite validates cryptographic key generation, signing, and transaction processing for *secp256k1*, *ed25519*, and *Dilithium*. Key highlights include:

- **Key Generation:** Confirms that secret and public keys adhere to expected sizes (e.g., 32 bytes for *secp256k1*, 2528 bytes for *Dilithium*).
- **Signing:** Validates that signatures are non-empty and match expected sizes (e.g., 64 bytes for *ed25519*, 2420 bytes for *Dilithium*).
- **Full Transaction Process:** Ensures end-to-end functionality for key generation, signing, and verification across all key types.
- **Deterministic Key Pair Generation:** Confirms that *Dilithium* key pairs derived from the same seed are consistent.

Performance and Throughput Comparisons

Performance Analysis: Two text files are also included in the repository called “diff_list.txt” and “diff_changes.txt” which contain the list of files modified and the exact changes made respectively.

- **Key Generation + Public Key Derivation:** *Dilithium* exhibits higher latency due to larger key sizes.
- **Key Pair Generation + Signing:** *Dilithium* shows significantly higher latency compared to *secp256k1* and *ed25519*.
- **Full Transaction Process:** *Dilithium*'s latency is the highest, reflecting the computational cost of post-quantum cryptography.

IV. Unresolved Aspects of PQC Integration

While significant progress has been made in integrating CRYSTALS-Dilithium into the codebase, certain aspects remain unclear due to the lack of specific requirements beyond cryptographic operations. These unresolved areas require further clarification and potential adjustments as the integration evolves.

1. Account Key Type for Dilithium

Current Status: It is unclear whether Dilithium should be introduced as a key type for account creation or limited to key pair generation, message signing, and signature verification. Keytype Dilithium has been added for the supporting keytypes for creation of accounts along with Dilithium.

Considerations:

- If Dilithium is used for account creation, additional changes to account management and validation logic may be needed.
- If limited to cryptographic operations, the existing account creation process can remain unchanged, with Dilithium used only for signing and verification.
- A new logic for account handling may need to be created if the logic for key operations does not work. This is not present even in the pq-CRYSTALS codebase.

2. ParseBase58 and Canonicality

Current Status: The functionality for *parseBase58* and canonicality checks has not been modified for Dilithium. These features remain untouched and are not recommended for use with Dilithium at this stage.

Reasoning: The original pq-CRYSTALS codebase does not currently support these functionalities. Implementing them without proper support could lead to inconsistencies or errors.

V. Conclusion

While Dilithium demonstrates higher latency and lower throughput, it provides quantum resistance, making it a critical addition for post-quantum security. The tests confirm that the integration of Dilithium maintains the robustness and consistency of the cryptographic framework. Future updates will focus on simplifying the build process and adding missing functionalities, such as canonicality checks and key parsing, as they become available.