Integration of Zero-Knowledge Proofs into the XRPL

This document presents the design, implementation, and evaluation of a framework for incorporating Zero-Knowledge Proofs (ZKPs) into the XRP Ledger (XRPL). We propose a novel approach that enables confidential transactions without sacrificing the ledger's performance or consensus guarantees. The implementation demonstrates the feasibility of ZKP integration, and the experimental results show that the proposed approach achieves strong privacy guarantees with minimal overhead. This work lays the foundation for privacy-enhanced transactions on the XRPL and provides insights for future research in scalable, privacy-preserving distributed ledgers. This is an informal report explaining the everything from adding libsnark (ZKP library) to evaluating results performed on the unit tests in the rippled code base. Here is the github link to the repository : Rippled_zkp

Contents

1 Overview of XRPL Architecture

The XRP Ledger (XRPL) is a decentralized blockchain that employs the Ripple Protocol Consensus Algorithm (RPCA) to validate transactions. It supports fast, low-cost payments, tokenization, and Decentralized Exchange (DEX) functionalities. Transactions are pseudonymous and transparent, with all account balances, transaction histories, and smart contract interactions immutably recorded on the public ledger. While this architecture ensures auditability and security, it lacks mechanisms to protect sensitive financial data, such as transaction amounts or participant identities.

The inherent transparency of XRPL poses challenges for enterprises and institutions that require confidentiality in financial transactions. Publicly visible transaction details can expose patterns, counterparties, and suppliers, potentially revealing sensitive business information.

2 Libsnark Integration

We have used a well-known library called Libsnark, which provides a zkSNARK implementation - a cryptographic method for proving/verifying in zero knowledge the integrity of computations.

2.1 Dependencies

The `libsnark` library relies on the following:

- C++ build environment
- CMake build infrastructure
- GMP for certain bit-integer arithmetic
- `libprocps` for reporting memory usage
- Fetched and compiled via Git submodules:
  - `libff` for finite fields and elliptic curves
  - `libfqfft` for fast polynomial evaluation and interpolation in various finite domains
  - Google Test (`GTest`) for unit tests
  - `ate-pairing` for the BN128 elliptic curve
  - `xbyak` just-in-time assembler, for the BN128 elliptic curve
  - Subset of `SUPERCOP` for crypto primitives needed by ADSNARK

`Libsnark` gives users the freedom to define their own circuits as well as implement a zkSNARK for R1CS secure in the generic group model (Groth16). It also gives users the choice of elliptic curves like Edwards (`ed25519`), `bn128` (default), `alt_bn128`, `MNT4`, and `MNT6`. There are also some custom options mentioned in the `CMakeLists.txt` according to a user's preference or OS.

3 Migrating Libsnark from C++14 to C++20

The goal was to integrate the `libsnark` cryptographic library into the `rippled` codebase, ensuring full compatibility with the C++20 standard. The original `libsnark` codebase was written for C++11/C++14, and several deprecated or removed features in C++17 and C++20 caused build failures. Additionally, platform-specific issues (notably on Apple Silicon/arm64) were encountered and addressed.

3.1 CMake and Build System Updates

- Updated the main project's `CMakeLists.txt` to set `CMAKE_CXX_STANDARD 20` and `CMAKE_CXX_STANDARD_REQUIRED ON`.
- Ensured that all Conan profiles and CMake toolchains used `compiler.cppstd=20` and `compiler.libcxx=libc++` (on macOS).
- For dependencies like `gRPC` that were not C++20 compatible, forced their build with `compiler.cppstd=14` via Conan settings.

3.2 Deprecated/Removed STL Features

- `std::bind1st` and `std::bind2nd`:
  - These were removed in C++17.
  - Replaced all usages with equivalent lambda expressions or `std::bind` where appropriate.
  - Example replacement:

```
// Old (C++14): std::bind1st(std::plus<int>(), 10);
// New (C++20): [&](int x) { return 10 + x; }
```

- `std::random_shuffle`:
  - Removed in C++17.
  - Replaced with `std::shuffle` and a random number generator.
  - Example replacement:

```
// Old: std::random_shuffle(vec.begin(), vec.end());
// New: std::shuffle(vec.begin(), vec.end(),
         std::mt19937{std::random_device{}()});
```

3.3 Library and Dependency Compatibility

- Ensured all dependencies (`libff`, `libfqfft`, GMP, OpenSSL, Abseil, gRPC) were built with the correct C++ standard and architecture flags.
- For OpenSSL, migrated to OpenSSL 3.x to ensure symbol compatibility with modern dependencies.
- For Abseil, disabled SSE4.1 optimizations on ARM by setting `absl_enable_sse4_1=Off` in Conan options.

3.4 Platform-Specific Fixes

- On MacOS/Apple Silicon (although not recommended for the time being), patched the build to avoid linking x86_64-only code and to use only portable C implementations where necessary.
- On Linux/x86_64, confirmed that all SIMD and assembly optimizations could be enabled for maximum performance.

4 Changes to Rippled (XRPL)

To enable shielded, privacy-preserving, and ZKP-based operations, we extended the Rippled codebase with a modular zero-knowledge protocol layer. Our design philosophy was to minimize disruption to the existing codebase, leveraging Rippled's robust primitives and transaction processing, while encapsulating all ZKP logic in a dedicated protocol subdirectory. The new logic draws inspiration from Zcash's architecture but is tailored specifically for XRPL's requirements.

4.1 Core Components of the ZKP System

The ZKP system introduces several key components to the Rippled codebase, each designed to support shielded transactions efficiently (Present in src/libxrpl/zkp):

4.1.1 Incremental Merkle Tree (`IncrementalMerkleTree.h`)

At the heart of the shielded pool is the *IncrementalMerkleTree*, an efficient, append-only Merkle tree with the following features:

- **Efficient Updates**: Supports $O(\log n)$ updates and batch appends.
- **Persistent Storage**: Allows serialization and deserialization for robust recovery.
- **Key Methods**:
  - `append`, `appendBatch`, `precomputeNodes`: Efficient leaf insertion.
  - `authPath`, `verify`: Generate and verify authentication paths.

The associated *MerkleWitness* struct encapsulates:

- A leaf, its authentication path, position, and root.
- Methods for verification and serialization, enabling succinct proofs of inclusion.

4.1.2 Shielded Value Transfers (`Note.h`)

Shielded value transfers are modeled by the *Note* structure, which includes:

- **Core Attributes**: Value, randomness, unique identifier, and paying key.
- **Key Methods**:
  - `commitment`, `nullifier`: Compute note commitment and nullifier using cryptographic hash functions.

  - `toBits`, `fromBits`: Convert notes to and from bit representations.
  - `random`, `createRandom`: Generate random notes.

- **AddressKeyPair Struct**: Represents a shielded address with:
  - Spending and viewing keys.
  - Methods for key derivation (`derivePublicKey`, `deriveViewingKey`) and note spending checks (`canSpend`).

4.2 Zero-Knowledge Proofs (`ZKProver.h`)

The *ZkProver* class encapsulates all ZKP generation and verification logic:

- **Circuit Management**:
  - `initialize`: Initialize circuits for deposit and withdrawal operations.
  - `generateKeys`, `saveKeys`, `loadKeys`: Manage cryptographic keys.
- **Proof Generation and Verification**:
  - `createDepositProof`, `createWithdrawalProof`: Generate proofs for deposit and withdrawal operations.
  - `verifyDepositProof`, `verifyWithdrawalProof`, `verifyProof`: Unified proof verification interface.
- **Utility Functions**:
  - `uint256ToBits`, `bitsToUint256`: Field and bit conversions.
  - `generateRandomUint256`: Generate random values for cryptographic operations.

4.3 New Transaction Types for Shielded Operations

To support shielded transactions, we introduced new transaction types, each extending the `Transactor` base class:

4.3.1 ZkDeposit (`ZkDeposit.h`)

Handles the deposit flow:

- **Validation**:
  - `preflight`: Checks required fields and proof size.
  - `preclaim`: Verifies cryptographic proofs and commitments.
- **Execution**:
  - `doApply`: Transfers XRP to the shielded pool and records the commitment.
- **Helpers**:
  - `createDepositProof`: Client-side proof generation.
  - Internal methods for pool management and proof verification.

4.3.2 ZkWithdraw (`ZkWithdraw.h`)

Handles the withdrawal flow:

- **Validation**:
  - `preflight`, `preclaim`: Ensure valid proofs, unique nullifiers, and valid Merkle roots.
- **Execution**:
  - `doApply`: Updates the shielded pool, prevents double-spending, and transfers XRP to the destination account.
- **Helpers**:
  - `createWithdrawalProof`: Proof generation for withdrawals.

4.3.3 ZkPayment (`ZkPayment.h`)

Supports shielded payments:

- **Validation**: `preflight`, `preclaim`.
- **Execution**: `doApply`.
- **Helper**: `verify_zk_proof` for payment proof verification.

4.4 Ledger Integration

The *STZKProof* class (`STZKProof.h`) extends `STBlob` to represent serialized ZK proofs within XRPL transactions. This enables:

- Embedding ZKP objects directly in the ledger.
- Compatibility with the new shielded transaction types.

The class provides constructors for various initialization scenarios and overrides type and textual representation methods for seamless protocol integration.

These changes to Rippled introduce a modular and efficient zero-knowledge protocol layer, enabling shielded transactions. By leveraging XRPL's robust primitives and encapsulating ZKP logic in dedicated modules, the system achieves privacy, scalability, and compatibility with minimal disruption to the existing codebase.

5 Improvements to the Current Approach

The original implementation of **libsnark** utilized a standard Merkle tree for managing commitments and generating proofs. Even thought this approach was functional, it had signification performance limitations due to $O(n)$ complexity of appending leaves and generating authentication paths. These inefficiencies became a bottleneck, particularly for large-scale applications requiring frequent updates and proof generation.

To address these challenges, we replaced the standard Merkle tree with an *Incremental Merkle Tree* (IMT). The IMT leverages $O(\log n)$ complexity for updates and proof generation, significantly improving performance. Key optimisations include:

- **Efficient Updates**: The IMT supports batch appends and precomputes nodes, reducing the overhead of frequent updates.
- **Persistent Storage**: Serialization and deserialization capabilities ensure robust recovery and state management.
- **Optimized Authentication Paths**: Cached sibling nodes enable faster generation of authentication paths, further enhancing efficiency.

In addition to the structural improvements, we optimized the zero-knowledge proof generation process by reducing the number of constraints in the arithmetic circuits. The primary performance gain came form eliminating redundant field element conversions for cryptographic values by bypassing the expensive packing/unpacking gadgets for values like *note_rho*, *note_r*, *note_a_pk*, *a_sk* and *vcm_r*. Instead, storing these 256 bit values directly as bit arrays without intermediate field element representations. This change alone removed approximately 1280 packing constraints (5 gadgets x 256 bits each).

These optimizations reduced the proof generation time from approximately 40 seconds to around 25 seconds and reducing memory allocation overheads, a significant improvement that enhances the practicality of ZKP-based transactions in real-world applications. The combination of the IMT and streamlined constraints ensures a scalable, efficient, and high-performance solution tailored to XRPL's requirements.

Additionally, the current implementation introduces proper empty hash handling with the *computeEmptyHash()* function that generates deterministic, level specific empty node values instead of using all zero hashes, which are a possible security vulnerability.

6 Testing

The integration of Zero-Knowledge Proofs (ZKPs) into the XRP Ledger (XRPL) was extensively tested to ensure correctness, performance, and compatibility with the existing system. This section describes the key test cases and their results, focusing on the functionality of the ZKP prover, ZKP transactions, and the performance comparison between standard Merkle trees and Incremental Merkle Trees (IMTs).

6.1 ZKP Prover Testing

The *ZKProver_test* suite was designed to validate the core functionality of the ZKP prover, including key generation, note creation, proof generation, and verification. The tests also evaluated edge cases, privacy guarantees, and the prevention of double-spending. The results confirmed the correctness and robustness of the ZKP prover, with all tests passing successfully. Below are the key components of the test suite:

- **Key Generation and Persistence:** The ZKP prover's ability to generate and persist cryptographic keys was tested. Keys were generated using the *generateKeys()* function, and their persistence was validated by saving and reloading them from a specified location. This ensures that the keys can be securely stored and reused across sessions.
- **Note Creation and Commitment:** The creation of ZKP notes, which represent shielded transactions, was tested using both random and manual methods. Each

note includes parameters such as value, randomness (ρ), and public key (a_{pk}). The correctness of note commitments, which are cryptographic hashes of the note parameters, was verified. Additionally, nullifiers, which prevent double-spending, were computed and validated.

- **Proof Generation and Verification:** The test suite evaluated the generation and verification of deposit and withdrawal proofs. Deposit proofs ensure that a note is correctly added to the shielded pool, while withdrawal proofs validate the spending of a note. The tests included:
  - **Deposit Proofs**: Random notes were created, and their deposit proofs were generated and verified. Debugging information, such as Merkle paths and nullifiers, was printed to ensure correctness.
  - **Withdrawal Proofs**: Notes were added to an IMT, and withdrawal proofs were generated using the authentication path and Merkle root. The proofs were verified to ensure that the note was valid and not double-spent.
- **Edge Cases:** The test suite included edge cases such as zero-value notes, large-value notes, and maximum allowable values. Privacy guarantees were validated by ensuring that different notes produced unique nullifiers, and the same note always produced the same nullifier. This prevents double-spending while maintaining anonymity.
- **Complete Workflow Testing:** A complete ZKP workflow was tested, simulating a real-world scenario:
  1. A user (e.g., Alice) creates a shielded note and generates a deposit proof.
  2. The note is added to the IMT, along with other dummy notes for anonymity.
  3. The user generates a withdrawal proof to spend the note, ensuring that the proof is valid and the nullifier is unique.
  4. Privacy was validated by ensuring that different notes produced different nullifiers, and double-spending was prevented by rejecting duplicate nullifiers.

6.2 ZKP Transaction Testing

The *ZKPTransaction_test* suite was designed to evaluate the performance of ZKP-based transactions, including cryptographic operations, ZKP proof generation, and Merkle tree operations. The suite also compared the performance of different cryptographic schemes (e.g., Secp256k1, Ed25519) and their integration with ZKP proofs. Below are the key components of the test suite:

- **Standalone Cryptographic Performance:** The performance of standalone cryptographic operations was tested for both Secp256k1 and Ed25519 schemes. Each test measured the time taken for key generation, signing, and verification over multiple iterations. The results showed that Ed25519 was faster than Secp256k1 for signing and verification, while key generation times were comparable.

- **ZKP Proof Performance:** The performance of ZKP deposit proofs was evaluated by generating and verifying proofs for multiple transactions. The results demonstrated that ZKP proof generation dominated the overall transaction time, accounting for 80-95% of the total time. Throughput was measured in transactions per second (tx/s), highlighting the scalability of the ZKP system.
- **Combined Cryptographic and ZKP Performance:** The suite tested the integration of cryptographic schemes with ZKP proofs, simulating real-world transactions. For example, a Secp256k1 signature was generated and verified alongside a ZKP deposit proof. The results showed that the combined performance was primarily constrained by the ZKP proof generation, with cryptographic operations adding minimal overhead.
- **Comprehensive Performance Comparison:** A comprehensive performance comparison was conducted between Secp256k1, Ed25519, and ZKP-based transactions. The results highlighted the following:
  - **Secp256k1**: Moderate performance with balanced key generation, signing, and verification times.
  - **Ed25519**: Faster signing and verification compared to Secp256k1, making it suitable for high-throughput applications.
  - **ZKP**: Dominated by proof generation time, but essential for privacy-preserving transactions.
- **Merkle Tree Performance:** The suite evaluated the performance of Merkle tree operations, including insertion, authentication path generation, and verification. Both incremental and regular Merkle trees were tested, with incremental trees demonstrating superior performance due to their $O(\log n)$ complexity for updates and proofs.

6.3 Standard Merkle Tree vs Incremental Merkle Tree

To evaluate the performance of the Incremental Merkle Tree (IMT) compared to a standard Merkle tree, a detailed performance comparison was conducted. The test focused on three core operations: append, authentication path generation, and verification.

- **Append Operation:** The append operation measures the time taken to add a new leaf to the tree. The IMT updates only the path from the leaf to the root, leveraging cached nodes and frontier optimizations, resulting in $O(\log n)$ complexity. In contrast, the standard Merkle tree rebuilds the entire tree from scratch, leading to $O(n)$ complexity. The results showed that the IMT was significantly faster, with speedups ranging from 200x for small trees (depth 8) to 250,000x for large trees (depth 20).
- **Authentication Path Generation:** Authentication path generation involves creating a cryptographic proof of leaf membership in the tree. The IMT retrieves cached sibling nodes, achieving $O(\log n)$ complexity, while the standard Merkle tree reconstructs the tree levels on demand, resulting in $O(n)$ complexity. The IMT consistently outperformed the standard implementation, with speedups of approximately 1.2x to 1.3x across all tested depths.

- **Verification Operation:**Verification checks whether an authentication path proves the membership of a leaf in the tree. Both implementations use the same algorithm for verification, resulting in similar performance ($O(\log n)$). Minor differences were observed due to cache locality effects, but the overall performance was nearly identical.

7 Results

The performance of Merkle tree operations was evaluated using both Incremental Merkle Trees (IMT) and standard Merkle trees. The results are presented in two graphs: a bar chart showing the breakdown of transaction time by process and a line chart illustrating the overall trend of transaction time with increasing tree depth.

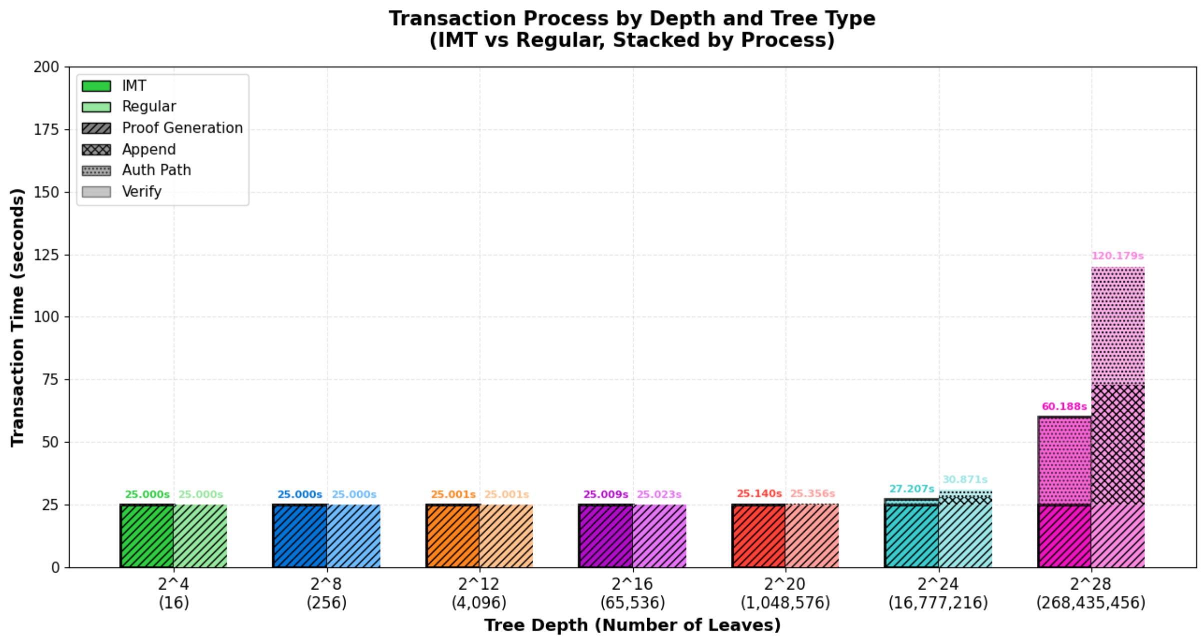

Figure 1: Transaction Process by Depth and Tree type

Figure 1 presents a bar chart that breaks down the transaction time into 4 key processes: *proof generation*, *append*, *authentication path generation*, and *proof verification*. The results are shown for both IMTs and standard Merkle trees across various tree depths, ranging from 2^4 (16 leaves) to 2^{28} (268,435,456 leaves).

- **IMT Performance**: The IMT consistently maintained a low and stable transaction time across all tree depths. The append operation, which benefits from the IMT's $O(\log n)$ complexity, was particularly efficient, with a constant time of approximately 25 seconds for smaller depths and a slight increase for larger depths.

- **Standard Merkle Tree Performance**: The standard Merkle tree exhibited significantly higher transaction times, especially for larger tree depths. The append operation's $O(n)$ complexity caused a dramatic increase in transaction time, reaching over 120 seconds for a tree depth of 2^{28}.

- **Process Breakdown**: For both tree types, proof generation dominated the transaction time, followed by authentication path generation and verification. However, the IMT's optimizations resulted in a much smaller contribution from proof generation compared to the standard Merkle tree.

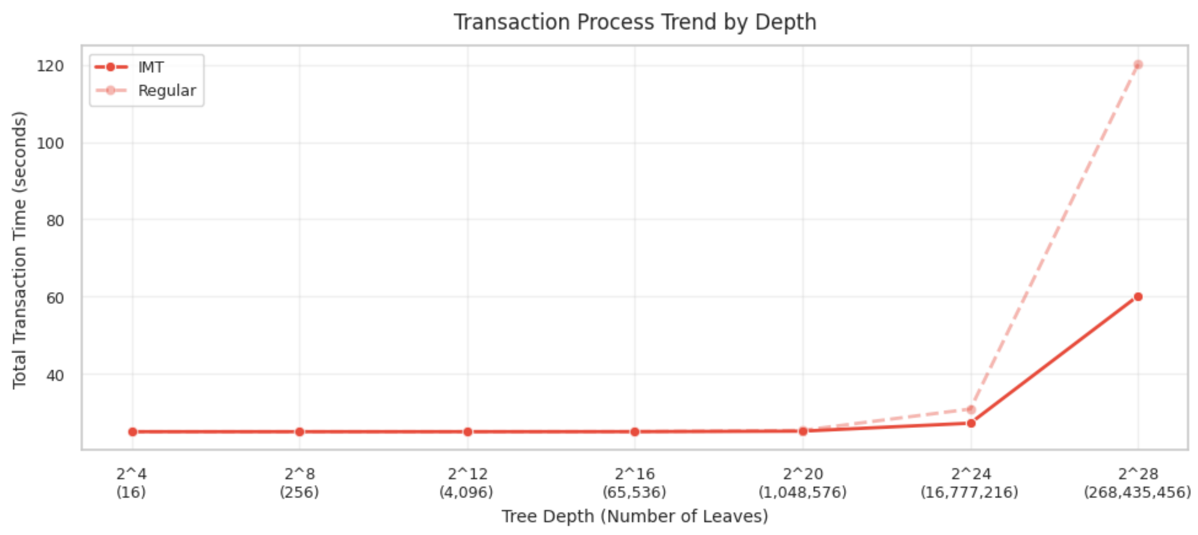

Figure 2: Transaction Process Trend by Depth (IMT vs. Regular)

Figure 2 illustrates the overall trend of transaction time as a function of tree depth. The line chart highlights the scalability of the IMT compared to the standard Merkle tree.

- **IMT Scalability**: The IMT demonstrated excellent scalability, with transaction times increasing only slightly as the tree depth grew. This confirms the efficiency of the IMT's $O(\log n)$ operations for append, authentication path generation, and verification.
- **Standard Merkle Tree Scalability**: The standard Merkle tree showed poor scalability, with transaction times increasing exponentially as the tree depth grew. This is attributed to the $O(n)$ complexity of the append operation, which becomes a bottleneck for large tree depths.
- **Performance Gap**: The performance gap between the IMT and the standard Merkle tree widened significantly for larger tree depths. At a depth of 2^{28}, the IMT's transaction time was approximately 60 seconds, compared to over 120 seconds for the standard Merkle tree.

The results demonstrate the superior performance and scalability of the Incremental Merkle Tree compared to the standard Merkle tree. The IMT's $O(\log n)$ complexity for append and proof generation operations ensures consistent performance even for large tree depths, making it a suitable choice for high-performance applications. In contrast, the standard Merkle tree's $O(n)$ complexity leads to exponential growth in transaction time, rendering it impractical for large-scale systems.

Considerations About the Implementation

Currently, the implementation is only recommended to run on Linux as it is in the ready-to-run state for that OS. We have tried to add conditions in the build commands to auto-detect the OS and build the respective packages accordingly. However, some packages do not sync well together with MacOS and Windows, as it is very difficult to find a combination of package versions that balance the OS (MacOS or Windows), *rippled*, and *libsnark* (C++20).

Even thought the tests were carried out by a superfast computer (Dell Alienware 18 with highest specs), it was still not enough to conduct tests for tree sizes 2^{32}, 2^{40} and so on.

We have integrated ZKPs as a different transaction type having different data structures (Incremental Merkle Tree) instead of the native SHAMap of XRPL. It was unsure so we went with the safe way of creating a new transaction type using ZKPs to ensure backward compatibility.

It is recommended to have `openssl@3` installed in the system, as using `1.1.1u` would lead to segmentation faults.