zkBridge Track Program

We cordially invite the community to join us in creating a community-driven extensible solution for ZKP-based bridges (zkBridge). The end goal of this track is to bring the community together to build end-to-end solutions for open-source zkBridge across different chains as public good and foster an open ecosystem towards building a secure, efficient, universal foundation for multichain interoperability. We hope this track in particular will help contribute diverse solutions for a defense-in-depth design for zkBridge with proof diversity and different implementations.

Introduction

zkBridge proposes a new solution for building trustless, permissionless, extensible, universal, and efficient cross-chain bridges using ZKP. With succinct proofs, zkBridge not only guarantees strong security without external assumptions, but also significantly reduces on-chain verification cost. zkBridge provides a modular design supporting a base layer with a standard API for smart contracts on one chain to obtain verified block headers from another chain using snarks. By separating the bridge base layer from application-specific logic, zkBridge makes it easy to enable additional applications on top of the bridge, including message passing, token transfer, etc.

Given zkBridge's modular design, the work needed for building a zkBridge is highly decomposable and parallelizable, making it a great project for community contribution and the hackathon. We have carefully designed the tasks in this zkBridge track such that different teams and participants can contribute to different components of a zkBridge which when put together can build high-quality solutions of zkBridges across different chains.

This track is co-hosted with zkCollective, an open community to help advance ZKP technologies in focused areas including zkBridge.

The overall goal for this effort is also to enable a defense-in-depth design, leveraging different, independent implementations and proof diversity; thus the overall solution combining the different implementations will provide even stronger security even if each implementation may have bugs. To achieve this goal, we encourage two parallel efforts: 1) developing different implementations of each component in a zkBridge from one network (or L2) to another; 2) developing a framework to combine the different implementations for defense-in-depth. Note that this framework could incorporate other non-zk approaches later as well such as an optimistic bridge, as part of a defense-in-depth solution.

Program Task Description

In short, this track aims to bring together the community and participants to build the foundation of a ZKP-based cross-chain bridge, as shown below.

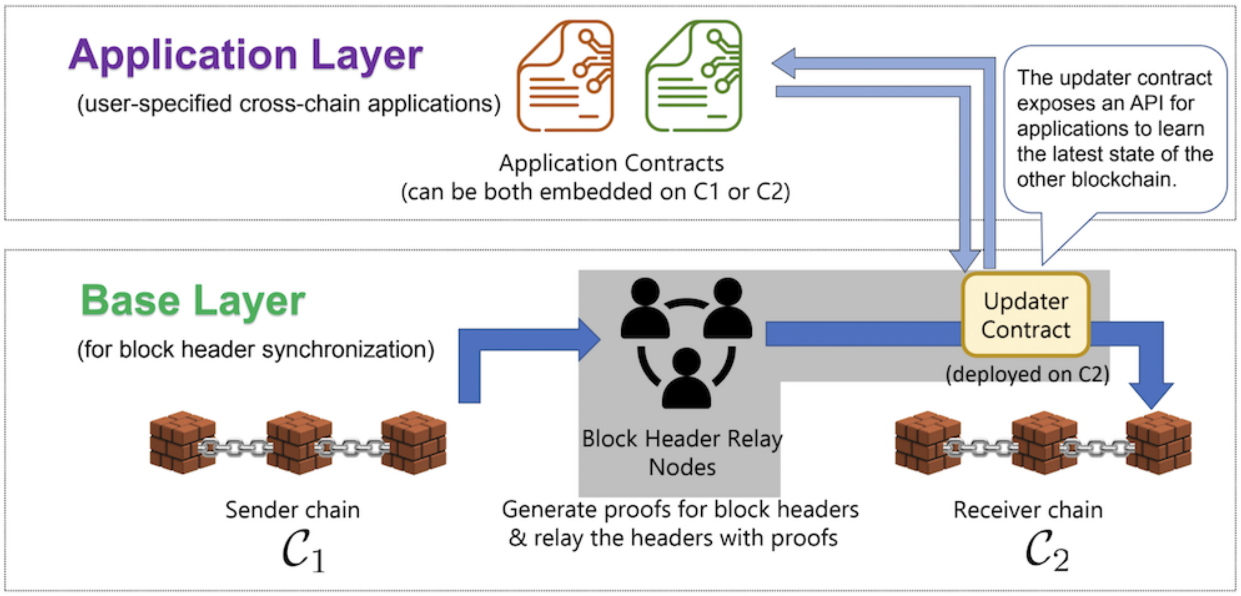
Application Layer
(user-specified cross-chain applications)
Application Contracts
(can be both embedded on C1 or C2)
The updater contract exposes an API for applications to learn the latest state of the other blockchain.
Base Layer
(for block header synchronization)
Updater Contract
(deployed on C2)
Block Header Relay Nodes
Sender chain
$\mathcal{C}_1$
Generate proofs for block headers & relay the headers with proofs
Receiver chain
$\mathcal{C}_2$

Figure 1. Overview of a zero-knowledge proof based bridge

The above graph shows the components of a typical Zero Knowledge (ZK) based bridge, including a proof system, light client, updater contracts, and DApps on top of the bridge.

We've created specific tasks and prizes for each component. The focus is on enabling inter-communication between popular L1/L2 chains, such as Ethereum, BSC, Polygon, Gnosis and Tendermint.

Category 1: Circuit

The community can greatly benefit from high-quality circuits that are easily incorporated into projects as standard libraries, particularly for popular code blocks like hash functions and elliptic curves. This category focuses on designated tasks that are building blocks for bridges on certain chains.

We recommend Circom and gnark as starting points for circuit programming languages since it's widely used and well-documented, but other circuit programming languages are also welcome.

Designated Task 1.1: Simple serialize (SSZ)

Simple serialize (SSZ) is the serialization method used on the Beacon Chain. SSZ is designed to be deterministic and also to Merkleize efficiently. SSZ relies on a schema that must be known in advance.

- Implement a circuit that implements the serialization and merkleization of SSZ.
- Existing implementation: Succinct Labs' SSZ for beacon chain

Designated Task 1.2: RLP serialization

RLP serialization is widely used by blockchains for serialization. RLP can be used to convert complex data structures to bytes so they can be consumed by a circuit. This task involves building a circuit that can check the correctness of RLP serialization. Specifically,

- Implement a circuit that checks input1 = RLP-encode(input2) (or equivalently input2=RLP-decode(input1)) for given inputs. Note the difficulty comes from the variable size of items in the data structure.
- Existing implementation: Yi Sun's RLP decoding

In the zkBridge track git repo, we provide an example implementation of RLP decoding circuits, as well as the instructions and test framework. Check the quick start for instructions.

Designated Task 1.3: Ecrecover for ECDSA

Ethereum and many EVM-compatible chains use the Elliptic Curve Digital Signature Algorithm (ECDSA) to validate the origin and integrity of messages. For example, BSC validators use ECDSA to sign block headers.

To validate a signature, EVM provides an ecrecover that recovers the public key (account address) of the signer from the signature and verifies the account address is the same as the claimed signer.

In this task, you will implement ecrecover in a circuit:

- Implement the circuit of ecrecover functionality.
- Related implementation: 0xPARC's ECDSA (missing ecrecover)

Designated Task 1.4: Ed25519 signature circuit

Ed25519 signature has been widely used as an CPU efficient signature scheme. In the blockchain world, it has been used in Tendermint to sign and verify the validator's signatures.

To validate a signature, you need to implement the verify circuit that takes the public key and signed message as input, outputs true if the signature is valid, otherwise throws an exception.

In this task, you will implement ed25519 verifier in a circuit

- Related implementation: https://github.com/Electron-Labs/ed25519-circom
- Bug warning: this related implementation has some bugs in the code, for example the following code:

  ```
  template fulladder() {
     signal input bit1;
     signal input bit2;
     signal input carry;

     signal output val;
     signal output carry_out;

     val <-- (bit1 + bit2 + carry) % 2;
     val * (val - 1) === 0;
     carry_out <-- (bit1 + bit2 + carry) \ 2;
     carry_out * (carry_out - 1) === 0;
  }
  ```

  The code doesn't check the constraint val == bit1 + bit2 + carry - 2 * carry_out

Designated Task 1.5: Amino serialization

Amino serialization is the key part of computing tendermint's blockhash. In this serialization, you are required to take Tendermint's block header as input and output the serialized byte string. Later, in the final task you will take these bytes into a hash function and calculate the final block hash.

- Related implementation: tendermint/go-amino: Protobuf3 with Interface support - Designed for blockchains (deterministic, upgradeable, fast, and compact) (github.com)
- Related doc: spec/encoding.md at master · tendermint/spec (github.com)
- Related data structure: spec/data_structures.md at master · tendermint/spec (github.com)

Designated Task 1.6: SHA256 hash

SHA256 is a popular hash function. Many block chains use it to calculate block hashes and transaction hashes. This task requires you to implement or audit the existing SHA256 hash function:

Task Details:

- Reference implementation: sha256 package - github.com/xuperchain/crypto/core/zkp/zk_snark/hash/sha256 - Go Packages
- Existing implementation: circomlib's sha236 circom circuits

Designated Task 1.7: Tendermint block header verification

Given a block header in Tendermint (cosmos), the light client shall be able to validate its integrity and authenticity. In short, the light client shall verify all ed25519 signatures from validators in the block header and handle validator changes.

Dependency:

Amino, ed25519, sha256

Task details:

- Implement the block header verification for Tendermint
- Related implementation:
- **Validator set update**
- Batched proofs and skip blocks policy.

Designated Task 1.8: Keccak-256

Keccak-256 is a popular cryptographic hash function from the SHA-3 family, commonly used in Web3. For instance, to sign a block header, BSC validators first hash the block header using Keccak-256.

Task Detail:

- Implement the complete Keccak-256 algorithm.
- Related implementation: Vocdoni's Keccak256 (missing multiple blocks).

Designated Task 1.9: Ethereum block header verification

Given a block header in Ethereum, the light client shall be able to validate its integrity and authenticity. In short, the light client shall calculate the hash of the block header encoded in SSZ (Simple Serialization), and validate that the corresponding BLS signatures are from the sync committee.

Task Detail:

- Implement the block header verification for Ethereum.
- Related implementation: Proof of Consensus for Ethereum.
- We should tell them to batch and skip blocks if the sync committee doesn't change.

    - Skip blocks: since sync committee only changes every 27 hours, we don't need to prove every blocks, we only need to prove following blocks:
        - User requested blocks: block that contains a cross-chain transaction
        - Sync committee update block: block that contains the next sync committee

Designated Task 1.10: Ethereum sync committee update

Sync committees are chosen every 256 epochs (~27 hours) consisting of at least 128 validators. The sync committee signs every block. The light client shall update the new sync committee every 256 epochs.

Task Detail:

- Implement Ethereum sync committee update.
- Related implementation: Proof of Consensus for Ethereum.

Designated Task 1.11: BSC single block header verification

Putting everything together, a BSC light client can be implemented. Given a block header of BSC (e.g. block 7705800), the light client will calculate the hash of the RLP-encoded block header, and verify that the ECDSA signature is from a legitimate validator.

Task Detail:

- Implement BSC block header verification.
- Related implementation: Darwinia's BSC Light Client.

Designated Task 1.12: BSC authority set update

BSC updates the authority set every Epoch, at the checkpoint header. For example, BSC block 7705800) includes the addresses of the 21 validators. Given a series of block headers, the light client shall extract validator addresses securely.

Task Detail:

- Implement the validator address extraction from multiple block headers.
- Related implementation: Darwinia's BSC Light Client.

Category 2: Smart Contracts

As shown in zkBridge, it uses an updater smart contract on one chain to verify and accept proofs of block headers of another chain from relay nodes. Figure 1 shows how the updater contract maintains a list of recent block headers and updates it after verifying the relay node proofs. The contract provides an application-agnostic API for smart contracts to access the latest block headers of the sender blockchain and build application-specific logic.

In this category, the participants are expected to implement the framework of updater smart contracts and the updater contracts for specific chains.

Designated task 2.1 Updater contract for Ethereum and Gnosis

- SSZ encoding of block header
- Batched proof generation/verification and skipping block policy
    - Batched proof generation aims to generate multiple block header proofs to reduce the proof verification cost. It will merge multiple block header verifications into one giant proof.
    - The Skipping Block Method is a feature of the light client. Since the sync committee changes every 27.3 hours, we do not need to verify intermediate unused blocks. We only need to verify two types of blocks:
        - Those requested by the user
        - Those that handle the sync committee change
- Sync committee set update and maintenance

Designated task 2.2 Updater contract for BSC

- Batched proof verification
- Validator set update and maintenance
- RLP decoding of block header
- Fork resolution

Designated task 2.3 Updater contract for Polygon

- Batched proof verification
- Validator set update and maintenance
- Fork resolution

Designated task 2.4 Updater contract for Tendermint (Cosmos)

- Batched proof verification / Skipping block policy
    - Similar to the Ethereum task, but the validator committee changes much more frequently. Fortunately, you don't need to update the committee frequently because most of the validator committee updates are minor updates and it will not change the verification result. You need to figure out the skipping policy (actually they have a document about this).
- Vadliator set update
- Fork resolution

Category 3: Block Header Relay Network

The zero-knowledge based bridge requires a relay network to deliver the block header securely from the source chain to the destination chain. A node in the block header relay network may connect to the full nodes of the source chain, and get the block headers continuously. Then the node generates the zero-knowledge proof of the block headers and delivers them to the updater contract on the target chain.

In this category, we assume the zero-knowledge proof generation is taken care of by the tasks in other categories, and use a placeholder for the zero-knowledge proof generation. We expect participants to build the block header relay network with the following components:

- Block header monitor
    - Given a source chain, build the monitor to continuously connect to full nodes of the chain to retrieve new block headers
    - Call the placeholder zero-knowledge proof generation api for the proof
- Updater contract pusher
- For different target chains, the pusher can call the api provided by the updater contracts to deliver the block headers and the zero-knowledge proofs

Designated task 3.1 Block header monitor for Ethereum

Designated task 3.2 Updater contractor pusher to Ethereum

Designated task 3.3 Block header monitor for BSC

Designated task 3.4 Updater contractor pusher to BSC

Designated task 3.5 Block header monitor for Polygon

Designated task 3.6 Updater contractor pusher to Polygon

Category 4: Message Relay Services

The off-chain message relaying node is a crucial component of a bridge as it delivers messages from the sender chain to the receiver chain. The relay node monitors a smart contract on the sender chain and gathers the messages for transmission. It generates a Merkle tree proof and delivers the messages to the receiver smart contract on the receiver chain.

Participants in this category are expected to build the message relay service, including the relay node and the relevant smart contracts. A relay service contains the following components:

- Source chain smart contract.
    - Interface to write binary messages.
- Receiver chain smart contract
    - Merkle Patricia tree proof verification
    - Interfaces for other applications to read data
    - User fee calculation algorithm
- Relay node

  - Monitor the source chain and get all the cross-chain transactions (block number/block hash, transaction index etc)
  - Generate transaction inclusion proof:
    - A Merkle Patricia tree proof of transaction
  - Submit the proof along with the transaction to the receiver chain smart contract

Designated task 4.1 Source chain smart contract for Ethereum

Designated task 4.2 Receiver chain smart contract for Ethereum

Designated task 4.3 Relay node for Ethereum to other chains

Designated task 4.4 Source chain smart contract for BSC

Designated task 4.5 Receiver chain smart contract for BSC

Designated task 4.6 Relay node for BSC to other chains

Designated task 4.7 Source chain smart contract for Polygon

Designated task 4.8 Receiver chain smart contract for Polygon

Designated task 4.9 Relay node for Polygon to other chains

Category 5: Applications on zkBridge

Cross-chain bridges have various use cases, including cross-chain token transfer/swap and NFT lock/stake, and can be used to transfer any message or share data across chains.

Participants are encouraged to develop innovative applications on top of the cross-chain bridge.

We provide following bridge interface (tentative):

On the sender chain side:

```
interface BridgeSendBytes {

    event BytesSent(bytes data);

    function sendBytes(bytes calldata data) external;

}
```

On the receiver chain side:

```
interface BridgeReceiveBytes {

    function receiveBytes(bytes calldata proof) external returns (bytes memory);
```

}

The sender chain side will take any byte string as input and emit an event containing the byte string. The app should keep monitoring the event. After the event is generated at some block, the developer should generate the transaction inclusion proof and send the proof bytes to the receiveBytes function. The detailed data in the proof will be different for different sending chains, we will provide detailed instruction soon. Then the app submits the proof to the receiver chain receiveBytes, it will verify the proof and output the transaction data.

Designated task 5.1 Token transfer

Transferring a token from one chain to another chain usually involves a lock and mint process. Oftentimes a wrapped token is minted on the target chain. You need to build a token transfer contract using our bridge interfaces.

Task Detail:

- Lock (receiverAddress, assetAmount, receiverChainID)
  - The user locks their asset using the lock contract, they must specify the receiver's address and chainID.
  - The Lock contract will emit an event: TokenLocked(receiver, assesAmount, chainID, UID)
- Mint (Proof)
  - The mint contract receives the verified block headers and the proof of a transaction inclusion about the TokenLocked event
  - Mint contract shall be able to decode the tuple (receiver, assesAmount, chainID, UID) from the proof.
  - The mint contract will make sure chainID is consistent with the current chain and the UID is never used.
  - The mint contract mints the asset to the user, assuming there is a liquidity pool that provides sufficient liquidity.

Designated task 5.2 Token swap

A cross-chain token swap is a process of exchanging one type of cryptocurrency token for another type of token on a different blockchain network. It often involves a sender and receiver smart contract, as well as the Dapp frontend for end users to use the swap service.

Task Detail:

- Sender
  - The Dapp shows the swap rate to the user, and calculates the gas and fees.
  - Once confirmed by the wallet, the sender contract receives the source assert, as well as the target token, the receiver's address and chain ID.
- Receiver

    - The receiver contract receives the verified block headers and the proof of transaction, and then mints the asset to the user, assuming there is a liquidity pool.
    - The receiver contract shall make sure each transaction executes only once.

Category 6: Defense in Depth

As mentioned earlier, it is important to develop a defense-in-depth solution, leveraging different, independent implementations and proof diversity, to achieve even stronger security even if each implementation may have bugs. In this category, participants are expected to design and develop a framework to combine the different implementations for defense-in-depth. Note that this framework could incorporate other non-zk approaches later as well such as an optimistic bridge, as part of a defense-in-depth solution. The participants are expected to provide a description of the design and its security analysis, and smart contracts to implement the framework, including the APIs for different bridge implementations to submit block headers (and provide the corresponding proofs and validation) as well as the logic for combining them to provide the API for the final defense-in-depth solution.

Category 7: Instantiation on XRPL

The XRP Ledger (XRPL) is an open source decentralized public blockchain. But differ from other blockchains that leverage smart contracts to build zkBridge, XRPL provides an alternative solution **Hooks** to support smart contract functionality on XRPL. Participants in this category are encouraged to use Hooks to develop zkBridge initiatives on XRPL.

Designated task 7.1 Generate proof with Hooks

zkBridge relies on the technology of Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (ZK-SNARK). Participants in this task are expected to integrate ZK-SNARK on XRPL and generate valid proof with Hooks:

- Verify the feasibility of building ZK-SNARK instances with Hooks, and investigate how to integrate proof circuits with Hooks.
- Create valid proof instances for transactions
- Hooks examples: Hooks (https://hooks.xrpl.org/) and some examples (https://hooks-builder.xrpl.org/develop)

Designated task 7.2 Verify the proof with Hooks

- For an incoming proof, accept a proof and verify its correctness
- Hooks examples: Hooks (https://hooks.xrpl.org/) and some examples (https://hooks-builder.xrpl.org/develop)

Designated task 7.3 Updater contract for XRPL

Suggested to proceed after task 4.1 and 4.2

- Similar to Category 2, perform batched proof verification
- Validator set update and maintenance

Designated task 7.4 Message Relay Service for XRPL

Total Prize: $10,000