Assignment 6

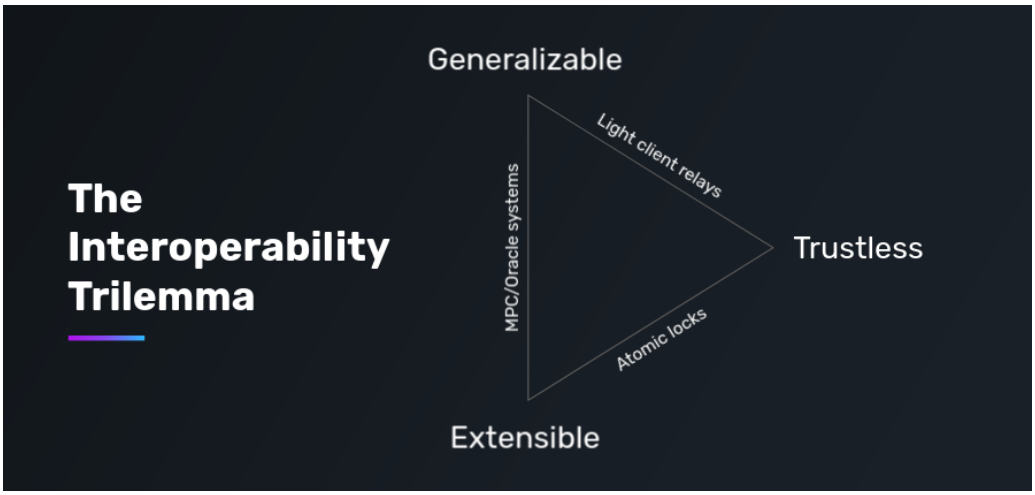
1. Interoperability trilemma

We talked before about scalability trilemma where L1 must sacrifice either decentralization, security or scalability. In a cross chain world there is also a famous interoperability trilemma.
Interoperability protocols can only have two of the following three properties:

- ***Trustlessness***
- ***Extensibility***
- ***Generalizability***

1. *Explain each aspect of the interoperability trilemma. Provide an example of a bridge protocol explaining which trade-offs on the trilemma the bridge makes.*

   [ANSWER]



- Trustlessness: maximizing the security of the system by disincentivizing validator collusion and making corruption attempts economically unfeasible.

  The bridge's security is equal to that of the underlying blockchain it is bridging. Outside of consensus-level attacks on the underlying blockchain, user funds cannot be lost or stolen.

- Extensibility: seamless extension of the underlying protocol to other domains.

  The selection of chains for both users and developers, as well as different levels of difficulty for integrating an additional destination chain.

- Generalizability: the protocol's capability of handling arbitrary cross-domain data.

> The capability of transferring information across multiple blockchains. This design enjoys strong network effects because of O(1) complexity — a single integration for a project gives it access to the entire ecosystem within the bridge. The drawback is that some designs usually trade off security and decentralization to get this scaling effect, which could have complex unintended consequences for the ecosystem.

State-of-the-art interoperability protocols are not able to offer all the above properties but are limited to choosing two at the tradeoff of excluding another one.

For example, Light clients & Relays bridges (for example: IBC, Near Rainbow) are strong with **Generalizability** because header relay systems could pass around any kind of data. They are also strong with **Truestlessness** because they do not require additional trust assumptions, although there is a liveness assumption because a relayer is still required to transmit the information. And do not need any capital lockup. These strengths come at the cost of **Extensibility**. For each chain pair, developers must deploy a new light client smart contract on both the source and destination chain, which is somewhere between O(LogN) and O(N) complexity depends on the underline chains. This protocol also introduces latency and speed drawbacks caused by synchronizing data and validating proofs.

2. ***[Bonus]*** *Are there any projects that focus on solving trilemma similarly like Ethereum solves the scalability issue? If yes describe how it solves the problem.*

   [ANSWER] Context and IBC (general bridge) are projects trying to solve this trilemma.

   Like Ethereum that adds scalability via L2/sharding as a layer on top of an existing secure and decentralized backbone, Connext establishes the two most important properties first that are essential to the longevity of the protocol. In the case of interoperability, the two most important properties are maximum Trustlessness and Extensibility. The Connext NXTP offers these two properties and is specifically designed to be usable on any chain while still being as secure as the underlying domains. Then Generalizability is added by plugging in natively verified protocols on top of NXTP. This adds a “Layer 2” to Connext’s interoperability network.

2. Ultra-light Clients

This week we focused on how to achieve ZK interoperability between chains. How we can use Zero Knowledge Proofs to sync a light client faster. This is especially useful for mobile first blockchains like Celo and is very useful for any blockchain.

1. *Describe a specification for a light client based on zero-knowledge proofs. You should explain at least how to get the client synced up to the current state of the*

blockchain. Preferably go as far as explain how transaction inclusion proofs are generated too.

[ANSWER]

Task

Light client that using ZKP to sync data and verify transaction inclusion.

Protocol

- **Initialization**
  - When the light client first connects to a full-node server of the chain, the server needs to send ZK proofs and chain status data (merkle roots of recipient, state, and transactions) to the client to validate the current state, this can be done in one of 2 ways:
    1. For every block, a ZK-Snark proof is generated to proof:
       - Block is well-formed.
       - Each transaction in this block is valid by checking the transaction signature
       - For POW block, the nonce is correct per difficulty.
    2. Use a recursive ZK-Snark proof (like Mina) for all blocks up to date, this proof proves:
       - Like 1 above, the current block is valid
       - The ZK-Snark proof for previous blocks is valid
  - The light client verifies these proofs, once passed, stores chain status locally. For both approaches above, the status data is merkle roots of historical merkle roots of different data (transaction, state, recipient). For example: use Merkle Mountain Range data structure to stores transaction merkle roots (in each block) and only save transaction MMR root in the light client.

  Note: since this is a non-interactive proof, any server can generate such proof and reused by other servers/clients after verification. This saves tremendous proof computation work.

- **Synchronization**

  We should not assume consistent connection between light client and network. After initialization, the later synchronization can be done periodically. Once connected, the client can ask server to only provide the proof of a specific block range. I.e., from say block 12345 to the latest. Similarly, this can be done in 2 ways,

1. For every block (from 12345 to the latest), a ZK-Snark proof is generated to proof.
2. Use a recursive ZK-Snark proof for all blocks up to date or a recursive ZK-Snark proof for a blocks range (from 12345 to the latest).

The light client then verifies and can update the local status (merkel roots of merkle roots) accordingly.

- **Transaction Inclusion Verification**

  To verify a transaction is included in a specific block by light client, a proof needs to be generated with following checks (public input: transaction data, transaction MMR root),

  - The transaction hash (computed from transaction data) is included in the transaction merkle root of a specific block.
  - This specific block is included in transaction MMR root.

2. *What is the relevance of light clients for bridge applications? How does it affect relayers?*

   [ANSWER]

   For bridge application, light client usually specifies the smart contract that deployed on one chain (destination) that receives status update, verifies the update is validate and stores new status from another chain (source) via relayer (connect to source chain). And vice versa.

   Again, each chain needs a relayer connect to it and act as full node or light client. The relayer receive block update from the source chain and call light client (smart contract) deployed in destination chain to update status.

   To illustrate, Harmony Horizon is a trustless bridge deploys light clients to facilitate cross-chain transactions. For example, in the token transfer use case, when a user locks their token on Ethereum, for minting the same amount of token on the harmony side, it requires Ethereum light client deployed on harmony to validate the lock transaction that happened on Ethereum. Similarly, when the user burns the minted token on harmony and tries to unlock the original tokens back to Ethereum, it requires the harmony light client deployed on Ethereum to validate the burn transaction that happened on harmony.

3. *Suppose code from Plumo is updated and was working in production on Celo. What would be the main difficulty in porting Plumo over to Harmony?*

[ANSWER] Celo and Harmony have a lot of common in underline technologies, specifically,

- POS based mechanism to select validators.
- Byzantine Fault Tolerance based consensus algorithm (Celo uses PBFT and Harmony uses FBFT).
- Leverage BLS multi-signature and need 2/3 validators' signature to validate a block.
- The validators won't change during a 24-hour's epoch interval. This also imply that the ultra-light client can efficiently pack all blocks in one epoch.

From migrating Plumo to work on Harmony perspective, I guess the main difficulty is to change Plumo's circuit to fit Harmony's block structure, namely:

- Support sharding and 250 validators for each shard.
- Reimplement crypto algorithm used by Harmony to create BLS multi-signature.
- Reimplement recursive zkSnark and make sure the proof size is acceptable.
- (in the future) MMR root calculation.

3. Horizon Bridge

Horizon is Harmony's bridge which allows crossing assets from Harmony to Ethereum/Binance and vice versa.

1. *Check out Horizon repository. Briefly explain how the bridge process works (mention all necessary steps).*

   [ANSWER] Copied from https://github.com/harmony-one/horizon

   Bridge Components

   - Used in Ethereum to Harmony flow
     - Bridge smart contract on Harmony
     - Ethereum Light Client (ELC) smart contract on Harmony
     - Ethereum Verifier (EVerifier) smart contract on Harmony
     - Ethereum Prover (EProver) is an Ethereum full node or a client that has access to a full node
     - Ethereum Relayer relays every Ethereum header information to ELC
   - Used in Harmony to Ethereum flow
     - Bridge smart contract on Ethereum
     - Harmony Light Client (HLC) smart contract on Ethereum
     - Harmony Verifier (HVerifier) smart contract on Ethereum
     - Harmony Prover (HProver) is a Harmony full node or a client that has access to a full node

- Harmony Relayer relays every checkpoint block header information to HLC

Ethereum to Harmony asset transfer

1. User locks ERC20 on Ethereum by transferring to bridge smart contract and obtains the hash of this transaction from blockchain
2. User sends the hash to EProver and receives proof-of-lock
3. User sends the proof-of-lock to bridge smart contract on Harmony
4. Bridge smart contract on Harmony invokes ELC and EVerifier to verify the proof-of-lock and mints HRC20 (equivalent amount)

Harmony to Ethereum asset redeem

1. User burns HRC20 on Harmony using Bridge smart contract and obtains the hash of this transaction from blockchain
2. User sends the hash to HProver and receives proof-of-burn
3. User sends the proof-of-burn to bridge smart contract on Ethereum
4. Bridge smart contract on Ethereum invokes HLC and HVerifier to verify the proof-of-burn and unlocks ERC20 (equivalent amount)

a) Comment the code for:

- *harmony light client*
- *ethereum light client*
- *token locker on harmony*
- *token locker on ethereum*
- *test contract*

Provide commented code in your submission.

[ANSWER]

https://github.com/geesimon/zku/blob/main/week6/HarmonyLightClient.sol

https://github.com/geesimon/zku/blob/main/week6/EthereumLightClient.sol

https://github.com/geesimon/zku/blob/main/week6/TokenLockerOnHarmony.sol

https://github.com/geesimon/zku/blob/main/week6/TokenLockerOnEthereum.sol

https://github.com/geesimon/zku/blob/main/week6/bridge.hmy.js

*b) Why HarmonyLightClient has **bytes32 mmrRoot** field and EthereumLightClient does not? (You will need to think of blockchain architecture to answer this)*

[ANSWER]

Ethereum is POW based, adding mmrRoot in EthereumLightClient to support transaction inclusion check, the best effort light client can do is to implement a FlyClient like mechanism. This requires replayer to send Log(N) (way better than SPV) of block headers to the smart contract and do verifications (each block will need to do a mmr root inclusion verification and block POW verification) inside the smart contract. Such verification (> 700 blocks) inside a smart contract is prohibitively expensive and could break the maximum transaction gas limit.

Harmony is POS based and a block is verified by checking BLS multi-signature. A good feature of BLS multi-signature is it can combine multiple messages and each message with different set of singers in one signature verification process. Leveraging such capability, the light client only needs to verify 1 signature every epoch (an epoch contains 32768 blocks or ~18 hours) and only needs the relayer sends 1 block per epoch. This makes the verification of signature inside a smart contract feasible. This also enable mmrRoot (for an epoch or a checkpoint) can be deployed to support faster transaction inclusion check.

Note: In Horizon, maximum of 16384 blocks can be handled in one update.

2. ***[Bonus]*** *What are checkpoint blocks? Do they differ from epoch blocks and how? Why are they used?*

   [ANSWER]

   Checkpoint blocks represent blocks inside an epoch. One checkpoint block can represent 1 to 16384 blocks.

   Unlike SPV that needs to verify and store every block header, HarmonyLightClient use checkpoint blocks to reduce verification cost and storage usage. See answer to 3.1.b on why HarmonyLightClient can combine multiple blocks in one checkpoint block update.

   Unlike epoch which represents fixed size of block (32768) and get generated every 18 hours, the underline block size for checkpoint block is adjustable. This gives control to trade-off between transaction speed (how long user needs to wait after token get burned in Harmony and can withdraw from Ethereum) and gas efficiency (how much bridge will cost to sync status).

3. ***[Infrastructure only]*** *Horizon still doesn't use zk-proofs to speed up light clients. What changes would you need to make to the code to apply initial state sync through zk-snarks? Provide pseudo code of improved version of light client.*

   [ANSWER] Similar to Plumo, we can generate a recursive zk-snarks proof from genesis block to the latest epoch, for each epoch the circuit (C1) need to prove:

- Aggregate public keys from header according to bitmap
- More than 2/3 validators signed, and the signature is valid
- Epoch number = previous epoch number + 1

Then to generate recursive zk-snarks, we need circuit (C2) to recursively compute the proof (from first epoch to latest epoch) to prove:

- Previous epoch proof is valid
- Current epoch is valid

The input parameter would be mmr root of genesis block, and output would be the new mmr root of last block in the latest epoch.

Then for the rest block (block generated after latest epoch), the relayer call prover to make a check point proof using similar circuit as C1. Then periodically, the relayer will call prover to make checkpoint proof and update mmr root.

Major functions in HarmonyLightClient.sol need to be changed as (pseudo code):

```
function initialize(
    bytes32 memory firstMMRRoot,
    Proof memory zkrProof,
) external initializer {
    MMRRoot = firstMMRRoot;

    require(verify(zkrProof, firstMMRRoot), "invalid recursive proof");

    MMRRoot = zkrProof.output["mmrRoot"];
}

function submitCheckpoint(Proof memory zkCheckPointProof)
                    external onlyRelayers whenNotPaused {
    require(verify(zkCheckPointProof, MMRRoot), "invalid recursive proof");

    MMRRoot = zkCheckPointProof.output["mmrRoot"];
}
```

4. Rainbow Bridge

1. ***[Infrastructure only]*** *Comment code implemented in [NearBridge.sol](). Note that they're using fraud proofs and not zk proofs.*

   [ANSWER]

https://github.com/geesimon/zku/blob/main/week6/NearBridge.sol

2. *Explain the differences between Rainbow bridge and Horizon bridge. Which approach would you take when building your own bridge (describe technology stack you would use)?*

   [ANSWER]

   Difference:

   - Trustless (Rainbow is Better)
     - Rainbow is a trustless bridge. To save gas cost, it uses Watchdog and incentive to encourage any user to verify a Near block. This constraint can be lifted after EIP665 is implemented in EVM.
     - Current implementation of Horizon depends on a trusted setup of Harmony relayer to post checkpoint to light client Ethereum smart contract.
   - Signature Verification (Horizon is supposed to be Better)
     - Harmony leverages BLS multi-sig to check signature of multiple validators. It enables the capability to verify all blocks in one epoch with one signature. Thus reduce the gas cost tremendously (although not implemented in current release) to verify signature inside Horizon.
     - Near network need one signature for each validator and is much less efficient. The validation of signatures in Rainbow cost about 500K gas.
   - Block Update on Ethereum (Rainbow is Better)
     - Harmony implemented a Merkle Mountain Tree based checkpoint block submission and the checkpoint size is adjustable. Checkpoint cannot cross epoch.
     - Since every Near header contains a root of the merkle tree computed from all headers before it. Rainbow leverages such capability to only post new block header during status update. This is more light and flexible.
   - *Others (Horizon is Better)*
     - Rainbow can only store 7 days of Ethereum block header in Near to storage usage. This requires user to mint token in Near after locked in Ethereum within 7 days. Otherwise, the transaction inclusion check will fail.

   RainBow and Horizon have more common than differences. They both have key component as: smart contract, relayer and prover. I like the extensibility of RainBow that allows more specific provers to be built to satisfy more use cases. Given it has better trustless design. I feel RainBow is better implemented.

   I guess I would take a hybrid approach by leveraging advantage of RainBow, Horizon and Plumo. Borrow the architecture from RainBow/Horizon and implement recursive Plumo like zkSnark for block/epoch verification.

3. ***[Bonus]*** *Explain how merkle mountain ranges work and how they can be used to do block inclusion proofs. (You can check FlyClient for a light client implementation that uses MMR).*

[ANSWER]

A Merkle mountain range (MMR) is a binary tree where each parent is the concatenated hash of its two children. The leaves at the bottom of the MMR are the hashes of the data. The MMR allows easy to add and proof of existence inside of the tree. MMR always tries to have the largest possible single binary tree, so in effect it is possible to have more than one binary tree. Every time you ask a Merkle root (the single Merkle proof of the whole MMR) you have the bag the peaks of the individual trees, or mountain peaks.

The unique structure of MMR makes it a better solution for inclusion proof than binary Merkle tree.

- MMR can be represented by array without need to fix the tree height.
- MMR is append-only, i.e., the value of any node is never changed. This means the history of the MMR roots is saved inside the tree.

Such structure allows efficient appends at the prover side and efficient block inclusion verifications at the verifier side. It also enables efficient subtree proofs, a proof that two MMRs agree on the first k leaves. The full node can prove that a transaction was included in the longest chain by just providing an MMR proof (to prove that a block belongs to the longest chain) in addition to the current transaction proof (which shows that the transaction is included in the block).

5. Thinking In ZK

1. *If you have a chance to meet with the people who built the above protocols what questions would you ask them?*

[ANSWER]

- Plumo
  - What challenge we could face if build a recursive zkSnark for Ethereum or BTC block verification proof?
  - Is it possible we can make a more generic architecture like:
    - One zk proof circuit for one chain that can be used by any other (smart contract capable) chains
    - One set of light client and prover for one chain that can verify and store any other chains' status update
    - One relayer for one chain that send status update to any other chains

What could be the major technical difficulties?

- Horizon
  - Why current implementation seems depend on a trusted setup of Harmony relayer?
- RainBow
  - Any new provers are implemented for specific use case?