# Assignment 6

## Q1 Interoperability trilemma

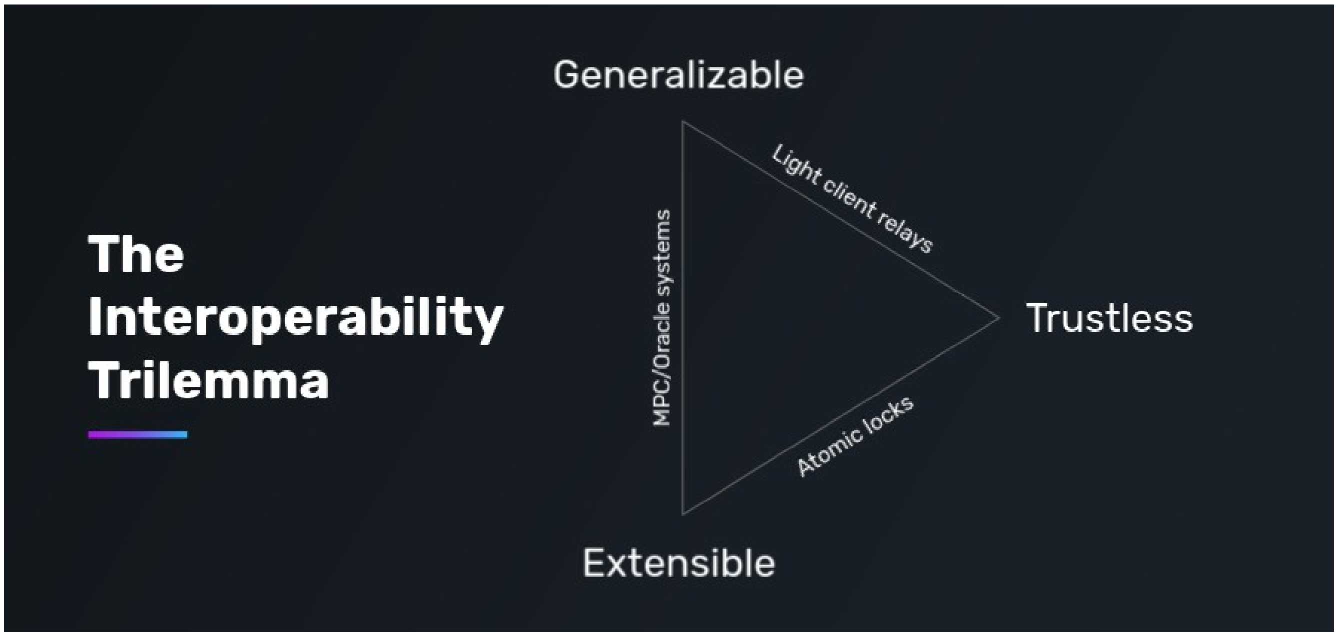
We talked before about scalability trilemma where L1 must sacrifice either decentralization, security or scalability. In a cross chain world there is also a famous interoperability trilemma.

Interoperability protocols can only have two of the following three properties:

- **Trustlessness**
- **Extensibility**
- **Generalizeability**

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 1.1:**



**Trustlessness:** Where the bridge security inherits the underlying chain security without relying on a different set of validators and without harnessing a user's overload of incentives for the validators, where users have to be insured up to the maximum ruggable amount

Examples: Hop bridge;

***Note:*** Aside from the fact that Hop bridge considered a locally verified system, It is not entirely trustless in the sense that it inherits all the L1 chain security assumption, it Hop adds some trust assumptions through their need for a fast arbitrary-messaging-bridge (AMB) in their system: the protocol unlocks bonder liquidity in 1 day rather than waiting a full 7 days when exiting rollups. The protocol also needs to rely on an externally verified bridge if no AMB exists for a given domain.

<u>Hop’s Trade-offs:</u>

- **<u>Generalizeability</u>:** It cannot support generalized data passing between chains permission-lessly on the contract level, for more specification, its possible for local verified systems to support generalized data passing between chains, by enabling cross-chains contarct calls but only if the function being called has some form of logical owner, as in the case of calling Uniswap functions for swappable tokens across chains, but in case of calling an NFT function across chains it’s not possible for the fact that the logical owner of the function “mint” on the destination chain should be the “lock” contract on the source chain, due to the special nature of NFT functions.

**Extensibility: able to be supported on any domain.**
**Generalizeability: capable of handling arbitrary cross-domain data.**

<u>Bridges interoperability trilemma:</u>

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 1.2:**

NO, A broad spectrum of consensus flavors & modularity schemes, also different market shares, make it a case by case approach, even some chains adopted ETH2.0 Roadmap and advances it like Harmony, I think all  of consensus flavors & modularity schemes strive like web 1.0 protocols and it would be on a dApp case by case to adopt each of them.

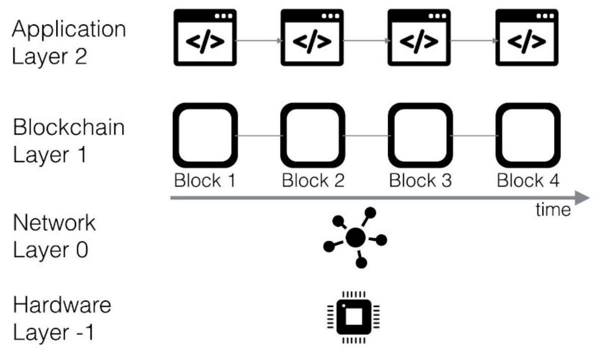
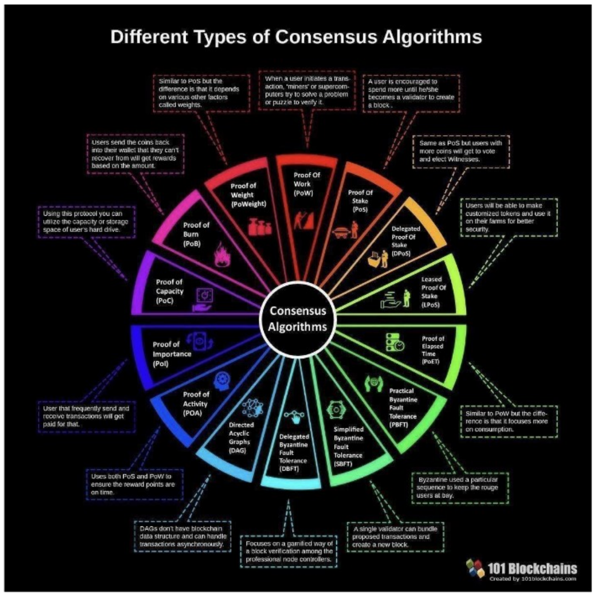


Figure 2: The different layers that a blockchain is constructed with. Note that layer 2 could also be another blockchain, for example an application specific blockchain.



Different Types of Consensus Algorithms
Consensus Algorithms
Proof Of Work (PoW)
When a user initiates a transaction, 'miners' or supercomputers try to solve a problem or puzzle to verify it.
Proof Of Stake (PoS)
A user is encouraged to spend more until he/she becomes a validator to create a block .
Delegated Proof Of Stake (DPoS)
Same as PoS but users with more coins will get to vote and elect Witnesses.
Leased Proof Of Stake (LPoS)
Users will be able to make customized tokens and use it on their farms for better security.
Proof of Elapsed Time (PoET)
Similar to PoW but the difference is that it focuses more on consumption.
Practical Byzantine Fault Tolerance (PBFT)
Byzantine used a particular sequence to keep the rouge users at bay.
Simplified Byzantine Fault Tolerance (SBFT)
A single validator can bundle proposed transactions and create a new block.
Delegated Byzantine Fault Tolerance (DBFT)
Focuses on a gamified way of a block verification among the professional node controllers.
Directed Acyclic Graphs (DAG)
DAGs don't have blockchain data structure and can handle transactions asynchronously.
Proof of Activity (POA)
Uses both PoS and PoW to ensure the reward points are on time.
Proof of Importance (PoI)
User that frequently send and receive transactions will get paid for that.
Proof of Capacity (PoC)
Using this protocol you can utilize the capacity or storage space of user's hard drive.
Proof of Burn (PoB)
Users send the coins back into their wallet that they can't recover from will get rewards based on the amount.
Proof of Weight (PoWeight)
Similar to PoS but the difference is that it depends on various other factors called weights.
101 Blockchains
Created by 101blockchains.com

## Q.2 Ultra light clients

This week we focused on how to achieve ZK interoperability between chains. How we can use Zero Knowledge Proofs in order to sync a light client faster. This is specially useful for mobile first blockchains like Celo, but 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 2.1:**

**Light clients allow a user to get sufficient evidence of a transaction's inclusion in a blockchain without having to operate a full node themselves.**

**First step: Bootstraping**

– Initiate(Blockchain) → (state0, proof): The client on an input of the genesis block B1, bootstraps its state state by running an interactive protocol with a full node and receives a proof of correct initialization.

**Second step: Updating**

– Update(state0) for (state1, proof): The client updates its state from state0 to state1 to reflect the newest view of Blockchain via an interactive protocol with a full node.

**Third step: Verifying**

– VerifyState(state0, state1, proof) : The client verifies proof that state1 is a correct transition from state0 and outputs b ∈ {0, 1}.

**Fourth step: Querying**

– Query(state, data) for (reply, proof): The client makes a query for data where data = tx
(e.g. timestamp or block height) or data = account (e.g. an account's address).

We also assume that data includes the type of query, i.e. current balance of an account, sender/receiver/value of a transaction, etc. The client receives a reply  and a proof . If data ∈ Blockchain.

**Fifth step: Verify_Querying**

– Verify(state, reply, proof) → b: The Client verifies proof for reply and outputs b ∈ {0, 1}.

**Sixth step: Submit transaction**

– Submit(state, tx, account, secret) for (state1) : Submit a transaction tx to Blockchain on behalf of account with secret information .

**How transaction inclusion proofs are generated too.**

**Check proof**

```
1: valid-proof $\rightarrow$ false
2: if (not.MmrRootList.Contain(rt)) then abort
3: for i $\in$ {0, N} do
4:     if (EpochNumber.Contain(ENi)) then abort
5:     else EpochNumberList.Insert(ENi) endif
6: endfor
```

**Update State**

```
7: valid-proof $\rightarrow$ Vrf(proof, MMRroot, state)
8: for i $\in$ {0, M} do
9:      MerkleTree.InsertLeaf($cmt_{newi}$)
10: endfor
```

**Query & Reply**

```
11: if ($V_{out}$ > 0), then SendValue($V_{out}$, Tx.Sender) endif
12: $rt_2$ $\rightarrow$MerkleTree.GetMerkleRoot()
13: for i $\in$ {0, M} do
14:     BroadcastCiphertext($C_i$)
15: endfor
```

**Broadcast transaction**

```
16: BroadcastMerkleRoot ($rt_2$) // Broadcast the new Merkle root
17: return
```

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

**Answer 2.2:**

**What is the relevance of light clients for bridge applications?**

In the case of compatibility in EVMs context, Ideally light clients should be implemented as a smart contract without the use of trusted oracles to achieve trust-less setup, As smart contracts to allow for native interoperability between independent ledgers, This would allow for verifying transactions of a blockchain A inside a contract of blockchain B.

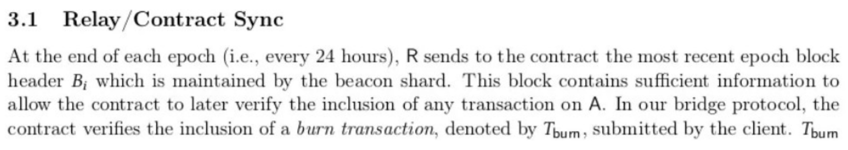
**Note:** Additional monetary costs for the contract's "gas" fees which can be potentially very high, As in the case of both Rainbow & Horizon bridges, due to the fact of differences between consensus algorithms used by the chains to be bridged.

**How does it affect relayers?**

Relayers (or relays): Ethereum2Harmony relay sends SPV-style block headers to ELC smart contract on Harmony, whereas Harmony2Ethereum relay sends checkpoint blocks to HLC smart contract on Ethereum. which implies in case of light client/smart contract **a time delay until it received the reply to its query**, Also incompatibility with blockchains without a smart contract VMs.

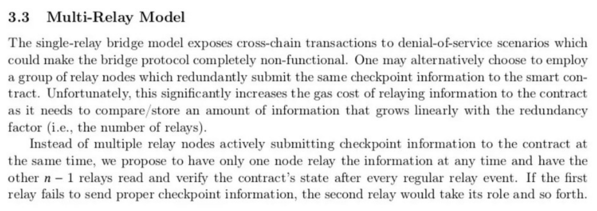
For the fact that Harmony is way faster in producing blocks than Ethereum, verifying harmony blocks on Etherum is prohibitively expensive, aside from Horizon's checkpointing technique used to decrease the cost of BLS veirfication, Relayers must wait to sync all light-client's contract transactions to attest to Harmony chain validators what transaction been burened $T_{burn}$ to unlock equivalent values.

So in conclusions relayers had to deal with very fast chain and it's networking complexities inherited from it's choice of BFT consensus algorithm to bridge assets to a slow PoW EVM compatible chain, and all that translates to a synchronization problem



**3.1 Relay/Contract Sync**

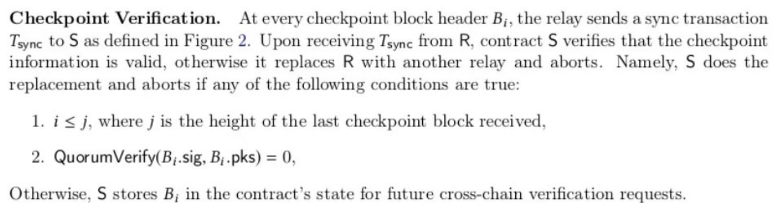
At the end of each epoch (i.e., every 24 hours), R sends to the contract the most recent epoch block header $B_i$ which is maintained by the beacon shard. This block contains sufficient information to allow the contract to later verify the inclusion of any transaction on A. In our bridge protocol, the contract verifies the inclusion of a *burn transaction*, denoted by $T_{\mathsf{burn}}$, submitted by the client. $T_{\mathsf{burn}}$



**3.3 Multi-Relay Model**

The single-relay bridge model exposes cross-chain transactions to denial-of-service scenarios which could make the bridge protocol completely non-functional. One may alternatively choose to employ a group of relay nodes which redundantly submit the same checkpoint information to the smart contract. Unfortunately, this significantly increases the gas cost of relaying information to the contract as it needs to compare/store an amount of information that grows linearly with the redundancy factor (i.e., the number of relays).

Instead of multiple relay nodes actively submitting checkpoint information to the contract at the same time, we propose to have only one node relay the information at any time and have the other $n-1$ relays read and verify the contract's state after every regular relay event. If the first relay fails to send proper checkpoint information, the second relay would take its role and so forth.



**Checkpoint Verification.** At every checkpoint block header $B_i$, the relay sends a sync transaction $T_{\mathsf{sync}}$ to S as defined in Figure 2. Upon receiving $T_{\mathsf{sync}}$ from R, contract S verifies that the checkpoint information is valid, otherwise it replaces R with another relay and aborts. Namely, S does the replacement and aborts if any of the following conditions are true:

1. $i \leq j$, where $j$ is the height of the last checkpoint block received,
2. $\mathsf{QuorumVerify}(B_i.\mathsf{sig}, B_i.\mathsf{pks}) = 0$,

Otherwise, S stores $B_i$ in the contract's state for future cross-chain verification requests.

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 2.3:**

The main difficulty for porting Plumo's prover to Harmony would be changing the underlying circuit to be verified with an instance of Harmony's EVM type verifier, without mentioning the solidity code complexity "Vharmony contract" to attest Harmony canonical chain Instance.

Generally the main difficulities would be the next two points, However a deep analysis of the underlying Celo's consensus algorithm implementation taken into consideration as it slightly differs from Harmony's (signature aggregation & block production rate), also Plumo tries to inflate it's own underlying cryptographic primitives security assumptions **(it's no more the same as the original authors of signature algorithms, compact signatures by dan boneh)** is another concern must but to test & benchmarking.

- Build SNARK-friendly hash-to-curve functions from established cryptographic primitives

- Present a concrete instantiation combining Bowe-Hopwood hash [Hop+19] and Blake2Xs [Aum+16] resulting in SNARK-friendly BLS signing

**<u>Analysis of Plumo's philosophy regarding consensus algorithms</u>**

**Plumo argument aganist Bitcoin's Nakamoto consensus:** Better scalability is attained by having, at any given time, a small committee of well-resourced validators who are in charge of agreeing on new blocks, using a Byzantine Fault Tolerance (BFT) consensus protocol

**Plumo's assumptions to validate Celo chain:** (Harmony can attest to these assumptions?)

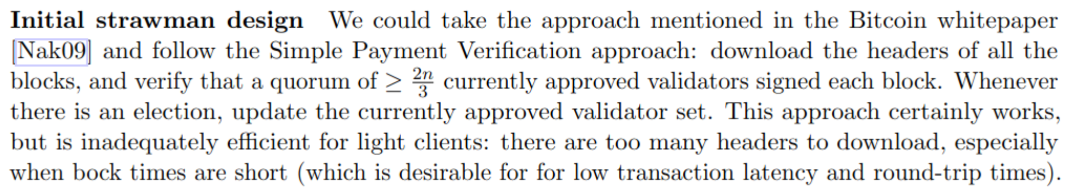
We assume a blockchain network where new blocks are chosen by BFT consensus executed by a set of validators, with some fixed target block time. We assume that at any time there are n validators. The set of validations can change (e.g., based on Proof-of-Stake elections) at the end of every epoch, which is some fixed number of blocks. Each validator has a public key signing pki (e.g., registered together with the proof-of-stake before elections occur), and signs blocks with the corresponding secret signing key.

- <u>Threat model:</u>

  - ✔ signatures by a quorum of ≥ 2n/3 validators is needed for a block to be considered valid (same as Harmony's threat model).
  - ✔ Any validator may be corrupted in subsequent epochs, after it has exited the validators set (i.e., during the epoch, validators' honest behavior is incentivized by slashing of its locked stake, but the lock is later lifted).
  - ✔ We assume that the adversary is computationally bounded, and that the cryptographic building blocks are indeed secure by their own definition.
  - ✔ SNARK scheme relies on a structured reference string, which we assume is securely generated by a suitable MPC setup ceremony

**Analysis of Plumo’s designing methodology as an ultra light client:**

- Epoch-based syncing



**Initial strawman design** We could take the approach mentioned in the Bitcoin whitepaper [Nak09] and follow the Simple Payment Verification approach: download the headers of all the blocks, and verify that a quorum of $\geq \frac{2n}{3}$ currently approved validators signed each block. Whenever there is an election, update the currently approved validator set. This approach certainly works, but is inadequately efficient for light clients: there are too many headers to download, especially when bock times are short (which is desirable for for low transaction latency and round-trip times).

**Note:** Plumo’s argument to not attest to PoW chain of EVM compatibility is low resource devices in feasibility to download all the block headers metadata and validate it, EVM block production don’t enjoy certain statistical properties to leverage this naive approach (i.e. difficulties changes), however BFT consensus based chains leverages epoch-time organization for changing validators sets, And so light clients can synchronize to only epoch-transition blocks until they reach the head of the chain, while trusting valdiators sets for within epoch blocks, Additionally, by following Ethereum state design, it’s enough to know the latest state root to make efficient light client queries.

- Signature aggregation

Every block needs to include enough validator signatures (a quorum) to convince a light client of its validity. Signature aggregation enables replacing all the signatures with a single signature. We use BLS signatures for signing individual block headers by ≥ 2n/3 validators. This allows us to use the non-interactive aggregation functionality of BLS signatures, having a single signature in a block instead of ≥ 2n/3 signatures. To achieve a threshold-like functionality, **we use an n-bit bitmap indicating which of the n validators contributed their signature to the aggregated multisignature. Since the public keys are known, this suffices for verification.**

- SNARK proofs for many epochs

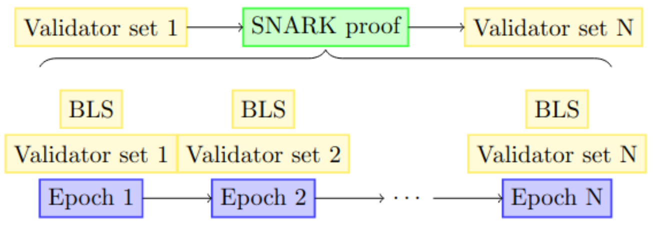


Figure 3.1: PLUMO architecture overview

**Idea: For porting Plumo & Celo**

**Constraints:** Assume: Harmony verifier contract: Celo_Vharmony.sol

1. **1- Create a solidity Implementation of Celo’s VM insides (Celo_Vharmony.sol)**
2. **2- A solidity Implementation of Hrmony verifier (Celo_Vharmony.sol) that can use Harmony’s VM opcodes to verify Celo’s :**

- The Celo blockchain uses the Istanbul IBFT consensus while Harmony leverage PBFT, IBFT to a class of BFT algorithms that assume a partially synchronous communication model, the set of validators in PBFT is static whereas IBFT features a dynamic validator set. This enables IBFT consensus mechanism based blockchains to confirm transactions faster than PBFT consensus mechanism based chains, Also Hrmony's underlying consensus algorithm is an updated version of PBFT (FBFT) it uses BLS (Boneh–Lynn–Shacham) multi-signature & RaptorQ fountain code to speed up the block broadcasting process and even if Harmony's FBFT speeds up to Celo's IBFT algo, the fact that IBFT allows for two types of nodes: validators that take part in the consensus protocol and standard nodes that validate blocks but do not participate in the consensus protocol.
- **Celo's IBFT algo** is deterministic, assumes a partially synchronous communication model, and guarantees safety independent of timing assumptions when n > [3f /2], where n and f are the number of total and byzantine nodes.
- **Harmony's FBFT** algo involves two major phases: the prepare phase and the commit phase. In the prepare phase, the leader broadcasts its proposal to all of the validators, who in turn broadcast their votes on the proposal to everyone else, The reason for the rebroadcasting to all validators is that the votes of each validator need to be counted by all other validators, The prepare phase finishes when more than 2f+1 consistent votes are seen, where f is the number of malicious validators, and the total number of validators plus the leader is 3f+1, The commit phase involves a similar vote counting process, and consensus is reached when 2f+1consistent votes are seen.

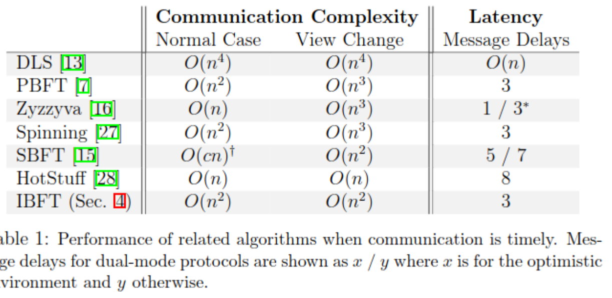
**Regarding the differences between Harmony's & Celo's Consensus Algorithm implementation**

| | Harmony | Celo |
|---|---|---|
| Consensus Algorithm | PBFT (Practical Byzantine Fault Tolerance) @ https://harmony.one/whitepaper.pdf<br>Note: I can only guess that Harmony's | The Istanbul BFT Consensus Algorithm @ https://arxiv.org/pdf/2002.03613.pdf |
| Signature-Scheme | BLS (Boneh–Lynn–Shacham) multi-signature @ https://www.iacr.org/archive/asiacrypt2001/22480516.pdf | ***The real Celo's implementation of signature aggregation is not document under peer reviewing.*** |
| Communication Complexity | Also, Table1 states that PBFT CC is O(n^3), worst case mode, however Harmony's underlying consensus algo. Is O(n), by inspiring O(1)-sized multi-signature of BLS scheme. | ***The real Celo's communication complexity (Validator-client) study is not document under peer reviewing and must undergo testing and bench-marking.*** |
| Latency | 3 | 3 |
| Liveness | N/A | N/A |
| Safety | N/A | N/A |
| epoch | 24 hours | 24 hours |
| Block time | 2 sec | 5 sec |

**Note:** Celo's compact signatures algorthim is slightly different from Boneh's

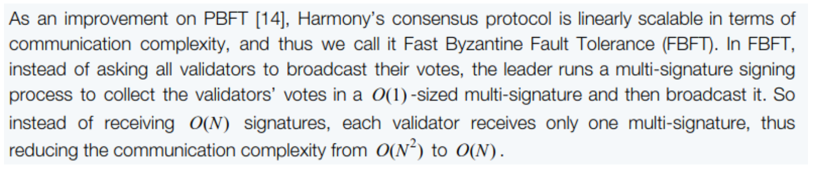
**Major challenges:**

**Plumo's circuit is build to prove to Celo's VM; which behind the fact it resembles Harmony's VM way of aggregating signatures , The absence of HVM pre-compile of the cryptographic primitives would be an issue.**

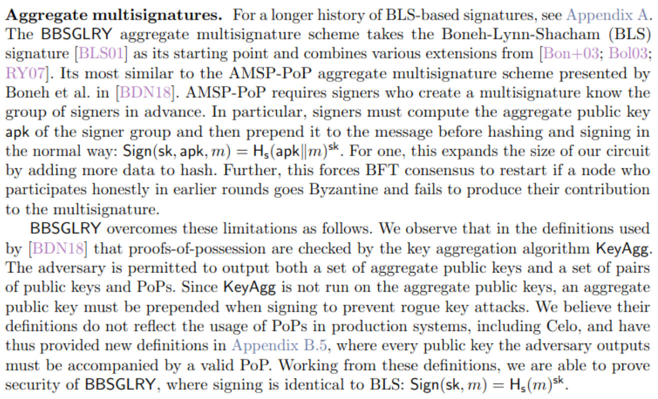


| | Communication Complexity | | Latency |
|---|---|---|---|
| | Normal Case | View Change | Message Delays |
| DLS [13] | $O(n^4)$ | $O(n^4)$ | $O(n)$ |
| PBFT [7] | $O(n^2)$ | $O(n^3)$ | 3 |
| Zyzzyva [16] | $O(n)$ | $O(n^3)$ | 1 / 3* |
| Spinning [27] | $O(n^2)$ | $O(n^3)$ | 3 |
| SBFT [15] | $O(cn)^\dagger$ | $O(n^2)$ | 5 / 7 |
| HotStuff [28] | $O(n)$ | $O(n)$ | 8 |
| IBFT (Sec. 4) | $O(n^2)$ | $O(n^2)$ | 3 |

Table 1: Performance of related algorithms when communication is timely. Message delays for dual-mode protocols are shown as $x$ / $y$ where $x$ is for the optimistic environment and $y$ otherwise.

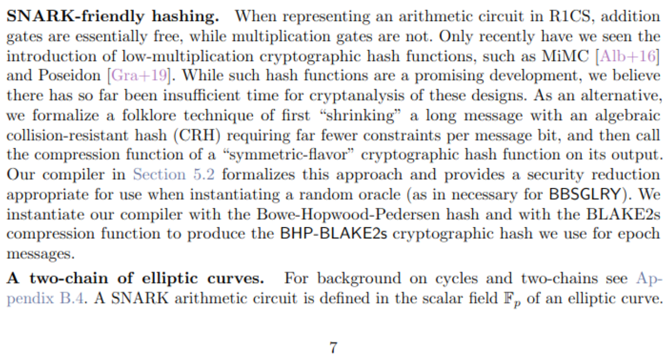


As an improvement on PBFT [14], Harmony's consensus protocol is linearly scalable in terms of communication complexity, and thus we call it Fast Byzantine Fault Tolerance (FBFT). In FBFT, instead of asking all validators to broadcast their votes, the leader runs a multi-signature signing process to collect the validators' votes in a $O(1)$-sized multi-signature and then broadcast it. So instead of receiving $O(N)$ signatures, each validator receives only one multi-signature, thus reducing the communication complexity from $O(N^2)$ to $O(N)$.



**Aggregate multisignatures.** For a longer history of BLS-based signatures, see Appendix A. The BBSGLRY aggregate multisignature scheme takes the Boneh-Lynn-Shacham (BLS) signature [BLS01] as its starting point and combines various extensions from [Bon+03; Bol03; RY07]. Its most similar to the AMSP-PoP aggregate multisignature scheme presented by Boneh et al. in [BDN18]. AMSP-PoP requires signers who create a multisignature know the group of signers in advance. In particular, signers must compute the aggregate public key apk of the signer group and then prepend it to the message before hashing and signing in the normal way: $\mathsf{Sign}(\mathsf{sk}, \mathsf{apk}, m) = \mathsf{H_s}(\mathsf{apk}\|m)^{\mathsf{sk}}$. For one, this expands the size of our circuit by adding more data to hash. Further, this forces BFT consensus to restart if a node who participates honestly in earlier rounds goes Byzantine and fails to produce their contribution to the multisignature.

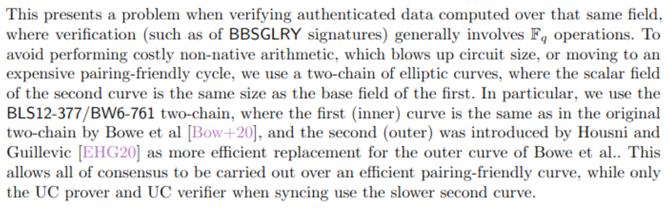
BBSGLRY overcomes these limitations as follows. We observe that in the definitions used by [BDN18] that proofs-of-possession are checked by the key aggregation algorithm KeyAgg. The adversary is permitted to output both a set of aggregate public keys and a set of pairs of public keys and PoPs. Since KeyAgg is not run on the aggregate public keys, an aggregate public key must be prepended when signing to prevent rogue key attacks. We believe their definitions do not reflect the usage of PoPs in production systems, including Celo, and have thus provided new definitions in Appendix B.5, where every public key the adversary outputs must be accompanied by a valid PoP. Working from these definitions, we are able to prove security of BBSGLRY, where signing is identical to BLS: $\mathsf{Sign}(\mathsf{sk}, m) = \mathsf{H_s}(m)^{\mathsf{sk}}$.



**SNARK-friendly hashing.** When representing an arithmetic circuit in R1CS, addition gates are essentially free, while multiplication gates are not. Only recently have we seen the introduction of low-multiplication cryptographic hash functions, such as MiMC [Alb+16] and Poseidon [Gra+19]. While such hash functions are a promising development, we believe there has so far been insufficient time for cryptanalysis of these designs. As an alternative, we formalize a folklore technique of first "shrinking" a long message with an algebraic collision-resistant hash (CRH) requiring far fewer constraints per message bit, and then call the compression function of a "symmetric-flavor" cryptographic hash function on its output. Our compiler in Section 5.2 formalizes this approach and provides a security reduction appropriate for use when instantiating a random oracle (as in necessary for BBSGLRY). We instantiate our compiler with the Bowe-Hopwood-Pedersen hash and with the BLAKE2s compression function to produce the BHP-BLAKE2s cryptographic hash we use for epoch messages.

**A two-chain of elliptic curves.** For background on cycles and two-chains see Appendix B.4. A SNARK arithmetic circuit is defined in the scalar field $\mathbb{F}_p$ of an elliptic curve.

7



This presents a problem when verifying authenticated data computed over that same field, where verification (such as of BBSGLRY signatures) generally involves $\mathbb{F}_q$ operations. To avoid performing costly non-native arithmetic, which blows up circuit size, or moving to an expensive pairing-friendly cycle, we use a two-chain of elliptic curves, where the scalar field of the second curve is the same size as the base field of the first. In particular, we use the BLS12-377/BW6-761 two-chain, where the first (inner) curve is the same as in the original two-chain by Bowe et al [Bow+20], and the second (outer) was introduced by Housni and Guillevic [EHG20] as more efficient replacement for the outer curve of Bowe et al.. This allows all of consensus to be carried out over an efficient pairing-friendly curve, while only the UC prover and UC verifier when syncing use the slower second curve.

**Celo’s VM cryptographic primitives that can verify:**

1. **BBSGLRY aggregate multi-signature scheme.**

Celo’s VM require validators to use BBSGLRY aggregate multi-signature scheme that’s an extension for Harmony’s BLS scheme used by validators that’s supposedly to run an instance of a Harmony VM verifier to verify BBSGLRY aggregation scheme, consequently the verifier solidity implementation must contains a struct of type BBSGLRY scheme that takes solidity HVM native types to construct.

**Note:** Celo’s validator/verifiers will verify Celo’s provers by checking PoP consensus algo run by Celo’s nodes {both the same}, consequently the compatiblity assumption of BLS: Sign(sk,m) = Hs(m)sk is dependent on

2. **SNARK-friendly hashing.**

For Celo’s VM to verify that the used hash function by R1CS circuit runs by Plumo, Harmony verifier running Celo’s VM instance must be able to verify the SNARK-friendly function that instatiate Celo’s compiler “Bowe-Hopwood-Pedersen hash” and also verifys the “BHP-BLAKE2s cryptographic hash” Celo uses for epoch messages.

As a consequence Harmony VM must implement a native hash type for A two-chain of elliptic curves, BLS12-377/BW6-761 two-chain, where the first (inner) curve is the same as in the original two-chain by Bowe et al [Bow+20], and the second (outer) was introduced by Housni and Guillevic [EHG20] as more efficient replacement for the outer curve of Bowe et al.

3. **Block syncing.**

Checkpoint solidity implementation to maintain fixed number of blocks per each chains epohc times Celo has 5s block times, this means transition proofs skip 17,280 blocks for every epoch message, one epoch, which is currently set to 24 hours worth of blocks.

In Harmony, the consensus and sharding process is orchestrated by the concept of epochs. An epoch is a predetermined time interval (e.g. 24 hours) Currently 1 epoch has 32768 blocks and with a 2s blocktime this is around 18.2 hours.

# Building blocks design

## BLS multi-signature scheme

Choosing a multisignature scheme. Multisignature schemes allow multiple parties to produce a single "multisignature" for a message m, attesting to the fact they each individually signed m.

**allowing us to separate verification time and signature size from committee size.**

While multisignatures for spans of epochs can be proved in a SNARK, when sending or checking for a transaction a client downloads the latest block header and verifies its multisignature directly.

**Plumo uses the BLS multisignature scheme [BLS01; BGS03; RY07] for both epoch messages and block headers, but instantiate the hash-to-curve function differently in the two settings, picking an optimized version for both outside and inside a SNARK**

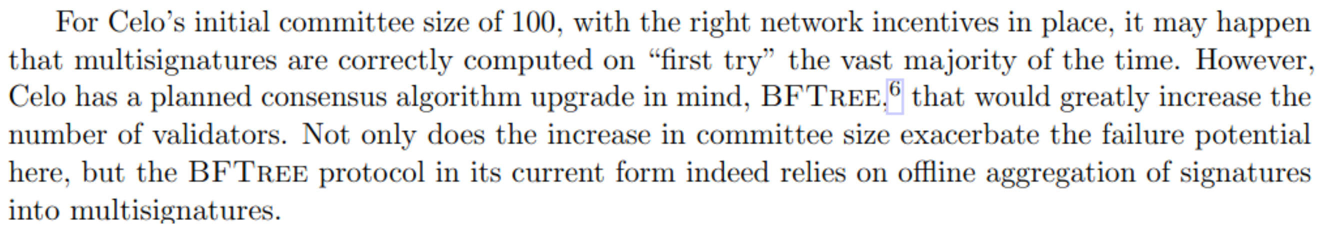
**BLS multisignature scheme**

| BLS multi-signature scheme | Source |
|---|---|
| BLS01 | D. Boneh, B. Lynn, and H. Shacham. "Short Signatures from the Weil Pairing". In: Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security. ASIACRYPT '01. 2001, pp. 514–532. |
| BGS03 | D. Boneh, C. Gentry, and H. Shacham. "Aggregate and Verifiably Encrypted Signatures from Bilinear Maps". In: Proceedings of the 22nd Annual International Conference on the Theory and Applications of Cryptographic Techniques. EUROCRYPT '03. 2003, pp. 416–432. |
| RY07 | T. Ristenpart and S. Yilek. "The Power of Proofs-of-Possession: Securing Multiparty Signatures against Rogue-Key Attacks". In: Proceedings of the 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques. EUROCRYPT '07. 2007, pp. 228–245 |

**BLS multisignatures have small signature and public key size (a single group element each) and fast verification (two pairings.**

**The most important property of BLS for Celo: (Must checked against Harmony)**

It supports offline aggregation of signatures into multisignatures, Interaction is not required, even each signer don't need to know the final set of signers in advance.


For Celo's initial committee size of 100, with the right network incentives in place, it may happen that multisignatures are correctly computed on "first try" the vast majority of the time. However, Celo has a planned consensus algorithm upgrade in mind, BFTree,[6] that would greatly increase the number of validators. Not only does the increase in committee size exacerbate the failure potential here, but the BFTree protocol in its current form indeed relies on offline aggregation of signatures into multisignatures.

Current status of Plumo's BLS scheme Instatiation to work for Celo: Even the BLS scheme doesn't require interactions for it work and aggregation can work offline, also the set of validators haven't to be known in advance, but the set of validators needs to be of constant size for each epoch. (Note: Must checked aganist Harmony's consensus setting).

| | | |
|---|---|---|
| BLS | Supports offline aggregation | Unknown final set of signers in advance (Non-interactive) |

**The increase in committee size exacerbate the failure potential here of reaching consensus finality over chain history not only epoch times (Note: A must check against Harmony).**

## Challenges of using BLS:

**There are two primary complications that come with our use of BLS signatures:**

**First:** They are set in bilinear groups and so we must use a pairing-friendly curve.

1. While pairing-friendly curves are less efficient at the same security level as their non-pairing-friendly counterparts, BLS signing, multisignature aggregation, and verification are still very fast.
2. The real problem follows from the fact that the **BLS verification equation cannot be efficiently proven in a SNARK over the same curve because of the mismatch between the field size and prime subgroup order.**

    - **While the same can be said for multisignatures schemes in the single group setting (Note: A must check whether Harmony BLS signatures secheme is in the single group setting or not).**

        ✔ *The difference is that is very easy to find efficient cycles of curves used to resolve this problem when they need not all be pairing-friendly. Contrarily, known cycles of pairing-friendly curves are very inefficient, and indeed there is some evidence suggesting more efficient pairing-friendly cycles may not exist.*

**Second:** Second, the security of BLS signatures holds in the random oracle model. **(While certainly there are valid theoretical concerns about the random oracle model)** , a more pertinent problem is that the "symmetric-flavor" cryptographic hash functions generally believed capable of securely instantiating random oracles using Fiat-Shamir transformation [FS86] are very costly to prove inside SNARKs.

So I can guess a solution to Plumo's BLS scheme was to build *A SNARK-friendly, composite algebraic-symmetric hash.*

## A SNARK-friendly, composite algebraic-symmetric hash.

Challenges of choosing a SNARK-friendly hash function for the BLS scheme:

**First:** we address the need to instantiate the random oracle in BLS and the apparent lack of SNARK-efficient options.

1. Solution to this problem is **a composite algebraic-symmetric hash function** that first uses **BHPedersen**, a SNARK-optimized variant [Hop+19] of the collision-resistant Pedersen hash [Ped92], to shrink the input to a single group element, and then calls **Blake2Xs10** [Aum+16] on that group element.

    - **Reason to call Blake2Xs10:** By calling Blake2Xs on a much smaller input, less iterations of its internal compression function are required, **which greatly increases prover efficiency.**

        - **Blake2Xs10 safety advantage:** We show that if Blake2Xs is safe to instantiate a random oracle with, then our composite hash function is also safe to instantiate a random oracle with under the discrete-logarithm assumption.

## A two-chain of pairing-friendly curves.

**As mentioned, BLS is positioned in the bilinear group setting.** This means to efficiently prove the correctness of a batch of signatures we need a second pairing-friendly curve with a prime order subgroup equal to the size of the base field of the first. While there exist cycles of pairing-friendly curves where the base field of each curve is equal to the prime order subgroup of the previous forming a closed loop, and thus allowing polynomial-depth recursive composition of proofs, **they are quite inefficient at reasonable security levels.**

**First:** at the 128-bit security level **(The most efficient known cycle - a two-cycle of MNT curves )**

The most efficient known cycle at the 128-bit security level is a two-cycle of MNT curves each defined over a 753 bit field[11], currently deployed in the Coda protocol [Mec]. The large size of these curves is less important for the SNARK prover that runs intermittently on a powerful server, and more impactful for the validators who must constantly make signatures over this large curve. Especially in terms of supporting signing on CPU and memory-constrained devices such as hardware wallets, **these large curves** presented a number of practical hurdles to an implementation that could keep up with our target block production time of 5 seconds.

- *Problems with this cycle (Used by coda)*

    1. *The large size of these curves is less important for the SNARK prover that runs intermittently on a powerful server ( Like Plumo's case).*
    2. *More impactful for the validators who must constantly make signatures over this large curve, Especially in terms of supporting signing on* ***CPU and memory-constrained devices.***
    3. *Celo's target block production time of 5 seconds.*

**(Note : A must check compatibility of curve cycle choosed for Harmony's 2 sec block production).**

- *Solution adopted by Plumo*

  *We resolved this problem by opting for a recently discovered **"two-chain" of curves** from [Bow+18]. While this pair of pairing-friendly curves only allows for a single level of recursion, the **first curve** is 377 bits, and has performance on-par with other pairing-friendly curves at the 128-bit level, **allowing for reasonable performance on hardware wallets**.*

  *The **second curve** is 782 bits. We split epoch periods up into fixed-size chunks to prove—so a bootstrapping client might get a first SNARK proving 256 days, a second proving 32 days, and a third proving 8 days to get them up to speed 296 days after the network launch. While polynomial-depth recursion would keep this at a constant single SNARK, modern SNARKs are so small and fast to verify that having to verify a few was still acceptable even for our most limited clients.*

  **(Note: Plumo protocol splits proving SNARK proves to 256 days, 32 days and 8 days, As a tweak to allow hardware constrained verifiers to verify the proof by the prover.)**

## Splitting the relation.

In the **BLS verification equation** the only computational work besides checking a pairing equation is computing a hash-to-curve function (the random oracle) on the message. Following cues from Zexe, we first opted to compute the entire composite hash function on the messages in a SNARK over the inner $E_{BLS}$ curve, and then prove it's correct verification over the larger, outer $E_{CP}$ curve in addition to proving correctness of the pairing equation

- **First:** compute the entire composite hash function on the messages in a SNARK over the inner $E_{BLS}$ curve.

- **Second:** prove it's correct verification over the larger, outer $E_{CP}$ curve in addition to proving correctness of the pairing equation.

(This turned out to be computationally more expensive overall for our prover)

**The reason they settled on implementing *the Groth16 proof system***;

Because it's a better-studied SNARK with a verification equation requiring: just **3 pairings**, plus a **linear number of exponentiations** in the size of the instance.

It turns out that doing the hashing in $E_{CP}$ (instead of splitting) the relation in this way is actually cheaper than the added cost of these exponentiations given the large size of epoch messages (Instance size).

**A known trick to reduce verifier time in Groth16**

1. Is to hash the instance down to a single field element, and modify the circuit to prove that the original computation is being performed on an opening of that hash of the correct form.
2. The verifier then receives the original instance, **hashes it themselves**, and uses that hash to compute the single exponentiation needed to verify the SNARK.

**Problem**: The problem with this method in our case is that the verifier is actually another SNARK prover, and again we run into the problem that *hashing inside SNARKs is expensive*.

Most important aspect that drives splitting protocol:
**[While only collision-resistance is needed here, using the more SNARK-friendly BHPedersen isn't feasible because it's not efficiently possible to use the same CRS of group elements in both curves.]**

Ultimate Plumo's splitting the relation in hashing between outer/inner curves protocol:

We found it most efficient **BHPedersen** over each message individually in $E_{CP}$ and to run **Blake2Xs10** over those messages in $E_{BLS}$. The result of hashing each epoch message with **BHPedersen** over $E_{CP}$ is interpreted as a string that we pack tightly into field elements over $E_{BLS}$.

This way we still do the more expensive part of the composite hash over the faster curve, but on a much smaller input, **minimizing the cost of checking the proof over the inner curve in the outer curve.**

## Final tweaks for a Plumo client.
**(for minimizing client verification time and data costs.)**

**Tip:** the client only needs to know the start and end epoch messages the SNARK is proving there exists a valid chain between.

1. We hash these two messages individually using Blake2Xs and **pack the 512-bit total result into a two field element instance**, minimizing the amount of exponentiation needed (Groth16 Trick).

**Tip**: having already confirmed the chain up to epoch a, receives a 392 byte Groth16 proof over $E_{CP}$ covering epoch a to b, the 95 byte instance, and the 4, 750 byte epoch message for epoch b.

2. They hash epoch messages and confirm they match the instance and then verify the Groth16 proof.

Note : As our SNARK circuits cover epoch chains of fixed powers two, it may require verifying multiple to get a client up to speed. **(A must check for Harmony's compatibility)**

**Note :** That a client needs only check the initial epoch hash of the first SNARK instance and the final epoch hash of the final SNARK. For the intermediate hashes the client only makes sure that the line up such that the second instance element of the first SNARK is the same as the first instance element of the second SNARK and so on.

## Bootstrapping the client

Needs only download the current epoch message (the genesis committee public keys **being hardcoded**) and a few SNARKs, which they batch verify. The result is a Plumo sync protocol which is extremely data and computation efficient.

We use a few different well-known batching techniques including a variant of the small exponent test [BGR98; CL06] to reduce 3n pairings to n + 2, computing multi exponentiations [Pip80], and computing a single final exponentiation after computing all Miller loops.

## Final Note: I am amazed by how far Plumo's developers went to build Celo's a specific-to-functionality light client, Aside from the fact that a lot of considerations must taken to insure underlying security assumption of the cryptographic primitives, SNARK setup and consensus rules.

## Q3. 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).

   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 3.1.a):**

Link: (harmony light client)
https://github.com/amrosaeed/zku/blob/main/Assignment6_Q3/HarmonyLightClient.sol

Link: (ethereum light client)
https://github.com/amrosaeed/zku/blob/main/Assignment6_Q3/EthereumLightClient.sol

Link: (token locker on harmony)
https://github.com/amrosaeed/zku/blob/main/Assignment6_Q3/TokenLockerOnHarmony.sol

Link: (token locker on ethereum)
https://github.com/amrosaeed/zku/blob/main/Assignment6_Q3/TokenLockerOnEthereum.sol

Link: (test contract) https://github.com/amrosaeed/zku/blob/main/Assignment6_Q3/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 3.1.b):**

HarmonyLightClient solidity implementation on ethereum chain resembles HVM supported by harmony validators at which they leverage a new mechanism for efficient proofs of transaction inclusion committing, A new data structure called a Merkle Mountain Range (MMR) is added to the block header, whose leaves are sequentially updated with the transaction root of each new block in all shards/cross-chain. Then with just the latest block header and a Merkle inclusion proof, an ultralight client can efficiently confirm any transaction, Generally these data structures must resembles the underlying chain participators artifacts, Where harmony infrastructure differs from Ethereum's, A consequence of harmony's adaption to shardning .

Indeed, contrary to Ethereum (single chain) Harmony contains a beacon chain and multiple shards, The beacon chain serves as the randomness beacon source and identity registerar, while the shard chains store separate blockchain states and process transactions concurrently.

For identity registration by the beacon chain of a transaction instantiated by an ultralight client in it's process of broadcasting to the leader validator  (prepare phase) a MMRroot leave must included in the blockheader as an inclusion proof of the transaction to be announced across different shards, at which a relayer can check "event Checkpoint()" "bytes32 receiptsRoot" parameter according to argument passed in transaction logs in ethereum chain.

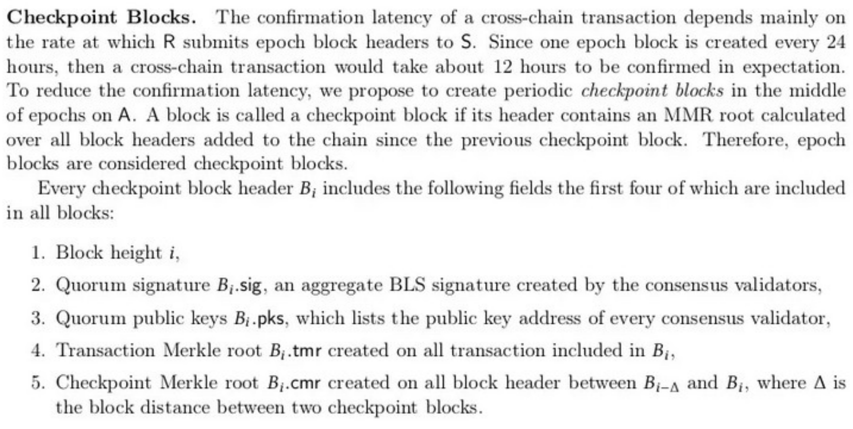
2. **[Bonus]** What are checkpoint blocks? Do they differ from epoch blocks and how? Why are they used?

**Answer 3.2):**

The check-pointing layer would be the one that pushes periodic checkpoints on the chain, once the checkpoints has achieved finality on the root chain, the transfers in all shards have been secured with harmony level security, the checkpoint has data like " stateRoot, transactionRoot, receiptsRoot, number, epoch, shard, time, mmrRoot and hash", they differ from epoch intervals as their functionality transcends to cross shards/chains while epoch blocks functionality are specific to certain shard/chain for preserving it's consensus rules.

**In Proof of Stake or BFT consensus blockchains for light clients to preserve security:**

The client also needs to verify account states and balances in the whole blockchain history, or consider the risk of long range attacks, BFT validators can join and leave, and a client needs to verify the consensus evolution through all validator signatures. A common technique to shorten the client's work is by storing intermediate checkpoints so that clients are not referring to the genesis block each time they verify the current validator set. On the other hand, validator set re-configurations, known as "epochs"



**Checkpoint Blocks.** The confirmation latency of a cross-chain transaction depends mainly on the rate at which R submits epoch block headers to S. Since one epoch block is created every 24 hours, then a cross-chain transaction would take about 12 hours to be confirmed in expectation. To reduce the confirmation latency, we propose to create periodic *checkpoint blocks* in the middle of epochs on A. A block is called a checkpoint block if its header contains an MMR root calculated over all block headers added to the chain since the previous checkpoint block. Therefore, epoch blocks are considered checkpoint blocks.

Every checkpoint block header $B_i$ includes the following fields the first four of which are included in all blocks:

1. Block height $i$,
2. Quorum signature $B_i.\mathsf{sig}$, an aggregate BLS signature created by the consensus validators,
3. Quorum public keys $B_i.\mathsf{pks}$, which lists the public key address of every consensus validator,
4. Transaction Merkle root $B_i.\mathsf{tmr}$ created on all transaction included in $B_i$,
5. Checkpoint Merkle root $B_i.\mathsf{cmr}$ created on all block header between $B_{i-\Delta}$ and $B_i$, where $\Delta$ is the block distance between two checkpoint blocks.

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

**Answer 3.3):** Mainly similar to **Answer 2.1**

**First step: Bootstraping**

– Initiate(Blockchain) → (state0, proof): The client on an input of the genesis block B1, bootstraps its state state by running an interactive protocol with a full node and receives a proof of correct initialization.

**Second step: Updating**

– Update(state0) for (state1, proof): The client updates its state from state0 to state1 to reflect the newest view of Blockchain via an interactive protocol with a full node.

**Third step: Verifying**

– VerifyState(state0, state1, proof) : The client verifies proof that state1 is a correct transition from state0 and outputs b ∈ {0, 1}.

**Fourth step: Querying**

– Query(state, data) for (reply, proof): The client makes a query for data where data = tx
(e.g. timestamp or block height) or data = account (e.g. an account’s address).

We also assume that data includes the type of query, i.e. current balance of an account, sender/receiver/value of a transaction, etc. The client receives a reply  and a proof . If data ∈ Blockchain.

**Fifth step: Verify_Querying**

– Verify(state, reply, proof) → b: The Client verifies proof for reply and outputs b ∈ {0, 1}.

**Sixth step: Submit transaction**

**– Submit(state, tx, account, secret) for (state1) : Submit a transaction tx to Blockchain on behalf of account with secret information .**

## Q4. Rainbow Bridge

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

**Answer 4.1:**

Link: https://github.com/amrosaeed/zku/blob/main/Assignment6_Q4/Q4.1/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 4.2:**

**The take way differences between Horizon & Rainbow bridges is the approaches they both adopts to address the prohibitive cost of transferring native assets (Harmony & Near) to Ethereum while maintaining bridge's trust-less features as they both share the same functionality advantage for transferring Ethereum assets to theirs.**

- **The key problem of Near's light client is the absence** Ed25519 signature available as an EVM precompile, So Verifying Near's block header requires expensive Ed25519 signature verification and it cannot be done as part of client's single unlock transactions.

- **The key problem of Harmony's light client in the absence** of BLS12-381 as an EVM precompile, So Verifying Harmony block header requires expensive BLS signature verification and it cannot be done as part of client's single unlock transactions.

**Horizon bridge key innovations to address relaying Harmony blocks to Ethereum are:**

- A checkpoint based Harmony light client on Ethereum that requires **only one BLS verification** for accepting the checkpoint block (1 block every x blocks, where 1 ≤ x ≤ 16384).

- A constant-size light client proofs of Harmony transactions which can be **verified on Ethereum in a gas effective manner.**

Harmony light client adopts a novel approach of MMR-based checkpointing for skipping frequent expensive pairing based verification.

**Note:** *I am not aware of the argument set by Horizon's developers about their approach of MMR-based checkpointing for skipping frequent expensive pairing based verification without however addressing long range attacks.* (Included as a question in Q5)

**Rainbow bridge key innovations to address relaying Near blocks to Ethereum are:**

NEAR uses Ed25519 to sign messages of the validators who approve the blocks, and this signature is not available as an EVM precompile. It makes verification of all signatures of a single NEAR header prohibitively expensive. So technically, we cannot verify one NEAR header within one contract call to NearOnEthClient. Therefore we adopt the optimistic approach where NearOnEthClient verifies everything in the NEAR header except the signatures. Then anyone can challenge a signature in a submitted header within a 4-hour challenge window.

This optimistic approach requires having a watchdog service that monitors submitted NEAR headers and challenges any headers with invalid signatures.

**Which approach would you take when building your own bridge (describe technology stack you would use)?**

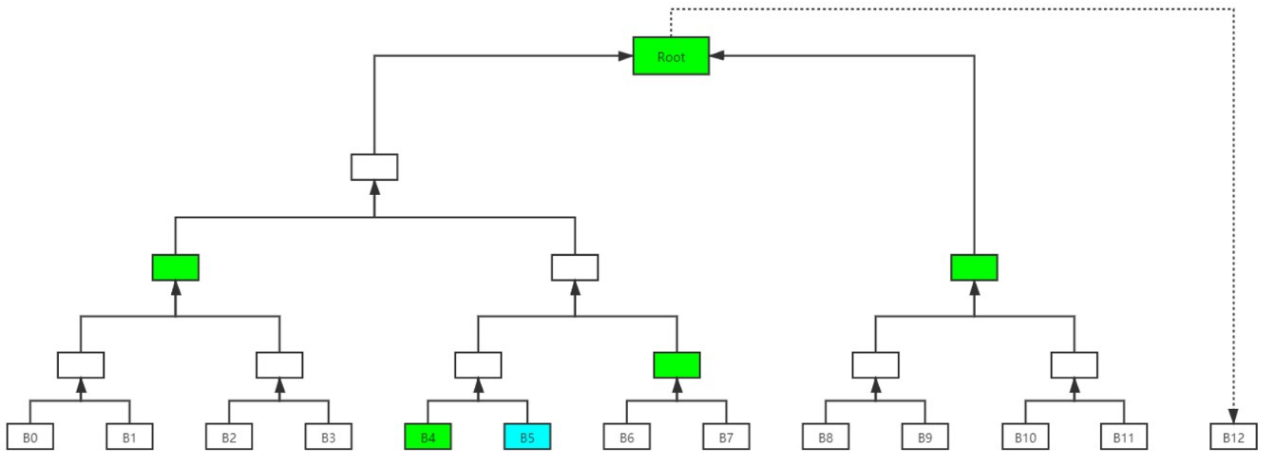
I'd prefer Horizon's approach, however from a security point of view I think they both nearly similar (Not sure), I'd prefer to hear from Horizon's developers how they addressed long range attacks for their checkpoiniting technique, So generally I think the problems with bridges mostly on the light clients side **as smart contracts are very resources constrained**, So apparently I'd go for Plumo's methodology and offload most of the load on the prover's side and maintaining the computation cost around 12-25$ (As they stated in the white paper), along with their way of doing the whole proving computation for 6 months in advance, but I may add an incentive systematization mechanism to insure that the prover entity choosed randomly.

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

**Answer 4.3:**

**How merkle mountain ranges work?**

Merkle Mountain Ranges are an alternative to Merkle trees. While the latter relies on perfectly balanced binary trees, the former can be seen either as list of perfectly balance binary trees or a single binary tree that would have been truncated from the top right. A Merkle Mountain Range (MMR) is strictly append-only: elements are added from the left to the right, adding a parent as soon as 2 children exist, filling up the range accordingly.



Contrarily to a Merkle tree, a MMR generally has no single root by construction so we need a method to compute one (otherwise it would defeat the purpose of using a hash tree). This process is called "bagging the peaks".

*Advantages of MMR over Merkle Trees:*

For the ordinary Merkle tree, Merkleroot needs to be recalculated for each newly added node. If there are a large number of nodes, this calculation will be huge, **MMR supports dynamically adding new nodes.**

**How MMR can be used in order to do block inclusion proofs. (FlyClient)**

FlyClient suggets a new Block Header, that contains one extra field namely the MMR root of the tree that commits the headers of all previous blocks, where it captures the block history commitments in an append-only data structure, also  that accumulated **difficulty** and accumulated **time** are included in each commitment to the MMR, meaning that each block header commits to 1) the sequence of all previous blocks and 2) total accumulated difficulty and time values.

By committing to difficulty and time data, difficulty transitions can be proven to have been executed correctly and verifiers can ensure that **no difficulty raising attacks have been executed.**

**The MMR root can replace the previous block hash and thus not increase the block headers size**. This requires a minimal change to the current block structure of Bitcoin and Ethereum, and can be implemented as a soft fork. LightClient will conduct only one additional check on the validity of the MMR root.

## Q5. 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?

**For horizon devs:**

I would like to ask, As I understands that checkpointing is skipping technique for verifying the whole range of BLS signatures, Is there a risk of long range attacks accompying this technique? Wiether it's was a strategy for transfering different harmony assets to ethereum?

**For Rainbow devs:**

How far their optimistic approach to transfer Near's assets to Ethereum can stand Impermanent-loss drawbacks?

**For FlyClient devs:**

How can their light client address selfish-mining problem?

**For Plumo devs:**

Is there any plans to adopt STARKs instead of SNARKs?, and how can this affect the underlying functionality and security for the light-client?