Zero-Knowledge University (ZKU): Assignment 6

Tal Derei

11 April 2022

Course Registration Email: Tad222@lehigh.edu

Discord Username: Margulus#4273

Github (Personal Repistory): `https://github.com/TalDerei`

Github (ZK Research): `https://github.com/TalDerei/Zero-Knowledge`

Q1. Interoperability Trilemma

1. Trustlessness, Extensibility, and Generalizeability:

The Interoperability Trilemma dictates that protocols can only achieve two of the following three properties at any given time: *Trustlessness, Extensibility, and Generalizeability.*[**1**] Trustless describes having equivalent security to the underlying domains on which the bridge operates on. Bridging protocols are only as secure as the external veifiers in their system. For example, locking up funds on Arbitrum/Optmism (L2 ZK-Rollups) shares the same security as the underlying Ethereum chain. If you decide to bridge those assets between rollups, your funds are now secured by the bridges external verifiers/validators instead. Secure bridges can achieve a level of trustless if their external validator sets are as hetergogenous/diverse and uncorruptable as the underlying chain on which the bridge is operating on. [**2**] Extensibility describes the property of being supported on any domain (i.e. bridge contract can be deployed on any chain and interact with any other chain). [**3**] Generalizability expands on extensibility, enabling bridges to handle arbitrary cross-domain data. This data can be a simple as a simple payment transaction beteween external parties, or complex interaction like a range proof proving that some value is within a specified ranges.

Solving the interoperability trilemma can be approached in the same way to tackling the scalbility trillemma. While Ethereum optimizes for security/decentralization at the cost of scalability, it tackles scalability via L2/sharding as a layer on top the core protocol. For example, NXTP is a locally verified system that's optimized for trustlesness and extensibility respectively, designed with shared security guarantees as the underling chain and useable on any domain. [BONUS] Generalizability is then added by plugging in additional protocols on top of NXTP.

Q2. Ultra Light Clients

1. Light Client Specification:

Light Clients Light clients don't need to download the full blockchain state, rather only block headers. Super light clients reduce this further by only requiring them to store some of the block headers, which is advantageous given that blockchains are expanding at an exponential rate. Unbounded state growth means blocks have larger headers and are producing blocks at faster block rates. Then the light clients check that all the headers are correctly signed by validators.

Specification for light-client based on zero-knowledge proofs is an example of **Plumo**, a zk-based ultralight client for the Celo blockchain (mobile-first blockchain) capable of quickly syncing to the latest network state using ZK proofs. The architecture is as follows: bridge contracts on blockchains A and B, relayers transfer information between chains, light clients that verify the validity of that information using proofs.

Light clients are able to quickly sync up to the current state of the blockchain because instead of downloading block headers and verifying their validity, light clients can us zk proofs (e.g. aggregating 6 months of transaction history into a proof that can be succinctly verified by a light client). After a client syncs up, they can perform merkle inclusion proofs in order to verify if users transactions has been in added in the heaviest chain. In that regard, light clients allow a user to get sufficient evidence of their transaction inclusion in a blockchain without having to operate a full node themselves.

2. Light Clients and Relayers

Relayers transfer information (block headers, messages, proofs of messages) between two blockchains A and B, and light clients verify the validity of those messages using block headers and proofs of messages.

3. Porting Plumo to Harmony?

The difficulty associated with porting the Plumo light client to Harmony's blockchain is Generalizibility-related bottlenecks. Light clients need to conform to the specification of the underlying chain on which they're operating on, which involves a new schema to transfer information between clients (client-to-client interaction) and information between client and the chain (client-to-chain interaction). This scales in complexity when you consider arbitrary cross-domain data that needs to be supported as well.

Q3. Horizon Bridge

1. Harmony-to-Ethereum

Codebase (Git Commit)

`https://github.com/TalDerei/ZKU/commit/af5c08783d984bd81512545bf63f4b24094b4564`

Horizon is a trustless Ethereum-to-Harmony bridge w/ cross-chain light clients + full nodes, relayers, and provers/verifiers.

Harmony to Ethereum:

1. User burns HRC20 tokens using functionality in bridge smart contract, receives transaction hash in return
2. User sends transaction hash to Hprover and recieves proof of burn
3. User sends proof of burn to brudge smart contract on Ethereum
4. Bridge smart contract on Ethereum invokes Hverifier to verify proof of burn and unlock ERC20

Ethereum to Harmony: 1. lock ERC20 tokens in bridge contract on Ethereum, and recieves transaction hash

2. user sends hash to Eprover and recieves proof of lock (block header info + path to lock tx)

3. user sends proof of lock to bridge smarrt contract on harmony blockchain

4. Bridge smart contract on Harmony mints HRC20 after invoking Everifier to verify proof of lock

Block headers on Harmony bridge include the '*bytes32 mmrRoot*' field since it's used to as a state commitment root to verify, for example, the aggregation of 6 months of transaction history into a proof that can be succinctly verified by a light client. Ethereum does not need this since it uses full nodes to do validation as normal.

Q4. Rainbow Bridge

1. Rainbow vs. Harmony Bridge:

The first obvious difference is that Rainbow needs to set an allowance to transfer XDAI from user to token locker, whearase this allowance doesn't exist on Harmony. After calling the token locker and emmiting an event, Raibowlib has to wait until the ETHonNEAR Client receives the Ethreum Header that contains the event, plus 25 more blocks to confirm the proper order. This another difference between the two protocols. Then, RainbowLib computes the proof of this event and submits it to the MintableFungibleToken contract, which then verifies that the proof is correct by calling ETHonNEARProver (which verifies the proof itself). MintableFungibleToken finally unpacks then Ethereum event and mints X amount of the native currency. The differences between the protocols is how the light client interaction works, the lockup period + allowance amounts.

Q5. Thinking in ZK

Assuming Ethereum's rollup centric-roadmap continues to develop at an exponential rate, cross-chain bridging will be important for bridging multiple L2-ZK rollups, something the space currently lacks. I can't, for example, move funds from Aztec to ZKSync, or zksync to hermez. Can ZK possibly help solve this issue, or making cross-chain transactions more secure across rollups? And if so, does implementing a zk system reduce the potential attack vectors and vunerabilities associated with moving funds between rollups.