

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

GRADUATION THESIS

**Building a Decentralized Bridge between Cosmos and
Ethereum using Zero-knowledge Proof (ZKP)**

TRẦN TRỌNG HIỆP

hiiep.tt190051@sis.hust.edu.vn

Specialization: Talent Computer Science

Supervisor: Associate Professor Nguyen Binh Minh Ph.D.

Signature

Department: Computer Science

School: School of Information and Communications Technology

HANOI, 08/2023

ACKNOWLEDGMENTS

I am deeply grateful to my lecturer, Associate Professor Nguyen Binh Minh, for his invaluable guidance and support throughout my journey in this project. His expertise and encouragement have been instrumental in shaping this work. I also extend my gratitude to Dr. Tran Vinh Duc, whose insights and guidance helped me in defining the direction of this research.

I would like to extend my heartfelt appreciation to all the members of TovChain company who generously shared their knowledge and offered assistance during the development of this project. Special thanks to the NFT, Orchai, and Ziden teams for their valuable contributions.

I am indebted to the BKCrypt0 team members, who have been my companions since the student computer olympic competition until now. Their camaraderie and support have made the journey more enjoyable.

I would like to express my thanks to the talented Bachelor K64 class for the wonderful four years we spent together at this school. Their unity and mutual support have helped us overcome challenges and grow together.

I am grateful to the teachers at the Hanoi University of Science and Technology for creating a conducive learning and working environment that fostered my academic growth.

Last but not least, I want to thank my family for their unwavering love, encouragement, and continuous support in all aspects of my life. Their belief in me has been a driving force in pursuing my goals.

ABSTRACT

Nowadays, with the rapid development of blockchain technology, it has garnered significant interest from major institutional investors and has had global impacts on governments, businesses, and individuals. Furthermore, the blockchain ecosystem has become increasingly heterogeneous, with a variety of blockchains co-existing. However, due to the lack of interoperability, these blockchains are unable to communicate with each other effectively. This necessitates the emergence of cross-chain bridges that facilitate the transfer of user transactions across different blockchains. Unfortunately, existing solutions for cross-chain communication suffer from either performance issues or rely heavily on trust assumptions of committees, which compromises the overall security. In this thesis, I propose a novel solution called zkBridge, which securely and decentralizes the bridging of transactions from Cosmos to Ethereum.

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION	1
1.1 Motivation	1
1.2 Objectives and Scope of the Graduation Thesis	1
1.3 Tentative Solution	2
1.4 Thesis Organization	2
CHAPTER 2. TECHNOLOGY BACKGROUND	4
2.1 Blockchain	4
2.1.1 Ethereum	4
2.1.2 Cosmos	7
2.1.3 Cosmwasm	17
2.2 Signature	17
2.2.1 ECDSA	17
2.2.2 EDDSA	18
2.3 Hash Funtion	18
2.3.1 SHA	20
2.3.2 Keccak256	22
2.3.3 Poseidon	22
2.4 Merkle Tree	22
2.4.1 AVL+ Tree	23
2.4.2 Fixex Merkle Tree	23
2.5 Zero-knowledge Technology	24
2.5.1 Defination	24

2.5.2 ZK-SNARK.....	25
2.5.3 Circom language.....	26
2.5.4 Zero-knowledge Proof (ZKP)	27
2.5.5 Constraint.....	29
CHAPTER 3. METHODOLOGY	32
3.1 Overview	32
3.2 Bridging Cosmos block headers	32
3.2.1 Bridge Cosmos block header mechanism.....	32
3.2.2 Evaluate the safety of the transferring Cosmos block header.....	33
3.2.3 Apply Zero-knowledge technology for transferring block header	34
3.3 Transferring deposit transactions.....	35
3.3.1 Transferring deposit transaction in a normal way.....	35
3.3.2 Transferring deposit transaction with deposit root mechanism	35
3.4 Claiming assets on Ethereum.....	37
3.5 Bridge validator set from Cosmos to Ethereum.....	38
CHAPTER 4. SYSTEM DESIGN	39
4.1 Architecture Overview	39
4.2 Cosmos contract system	39
4.3 Ethereum bridge system	41
4.3.1 Updater contracts	41
4.3.2 Bridge contract	43
4.3.3 Verifier contracts	45
4.3.4 Address management contract	45

4.4	Server	46
4.4.1	Zk bridge system api	48
4.5	Client	49
4.5.1	Functional Overview	49
4.5.2	Functional description	54
4.5.3	Non-functional requirement	55
CHAPTER 5. EXPERIMENT AND EVALUATION		56
5.1	Detailed design.....	56
5.1.1	User interface design.....	56
5.1.2	Database design.....	57
5.2	Application Building.....	59
5.2.1	Achievement.....	60
5.2.2	Illustration of main functions	61
5.3	Deployment	62
CHAPTER 6. MAIN CONTRIBUTIONS		63
6.1	The circuit for generating a proof for updating the new Cosmos block header on Ethereum is too large.....	63
6.1.1	Problem	63
6.1.2	Solution	63
6.1.3	Result	64
6.2	The information from Cosmos network is not compatible for Circom language	64
6.2.1	Problem	64
6.2.2	Solution	64
6.2.3	Result	65

CHAPTER 7. CONCLUSION AND FUTURE WORK	66
7.1 Conclusion.....	66
7.2 Future Work.....	67
REFERENCE	69
APPENDIX	71
A. THE FUNCTIONS IN COSMOS BRIDGE SYSTEM	71
A.1 Support token pair	71
A.2 Receive ew20 token	71
A.3 Update deposit tree	72
A.3.1 Update deposit tree mechanism	72
A.3.2 Withdraw ew20 token	73
B. UPDATER CONTRACTS IN ETHEREUM BRIDGE SYSTEM.....	74
B.1 Gate contract	74
B.2 Cosmos block header contract.....	75
B.3 Cosmos validator contract.....	76
C. ZKBridge Circuit.....	78
C.1 Verify new deposit tree root on Cosmos Bridge circuit	79
C.2 Verify new Cosmos block header circuit.....	82
C.2.1 Verify new deposit tree on Ethereum bridge.....	85
C.2.2 Verify claim transaction	85
D. SERVER API.....	86
D.1 Deposit transaction api	86
D.2 Token pair api.....	87

LIST OF FIGURES

Figure 2.1	This is an illustration of the block header for block 11855600 in the Oraichain network, which is a network built using the Cosmos SDK	9
Figure 2.2	The process of creating blockHash by the fields in block header . .	10
Figure 2.3	This is a representation of the transaction data within block 11855600 in the Oraichain network.	12
Figure 2.4	The process of calculating transactionHash and dataHash from the transactions data	13
Figure 2.5	This depicts the information of a validator within the validator set of block 11855600 in the Oraichain network.	14
Figure 2.6	The process of calculating validatorHash from the validator set . . .	15
Figure 2.7	This demonstrate the information of a validator's signature of block 11855600 in the Oraichain network.	16
Figure 2.8	This depicts the progress generate ZK proof.	28
Figure 2.9	Example of verify proof function in contract.	30
Figure 3.1	The decentralized mechanism facilitates the transfer of block headers from one chain to another	33
Figure 3.2	Constraints between attributes used for verifying the block header .	33
Figure 3.3	The decentralized mechanism facilitates the transfer of block headers from one chain to another with Zero-knowledge technology	34
Figure 3.4	Deposit root mechanism	36
Figure 3.5	Clamming asset mechanism	38
Figure 4.1	Architecture overview.	39
Figure 4.2	Cosmos contract system overview.	40
Figure 4.3	Ethereum bridge system overview.	41
Figure 4.4	Updater contracts overview.	42
Figure 4.5	Claim transaction flow.	43
Figure 4.6	Bridge contract function.	44
Figure 4.7	Verifier contracts function.	45
Figure 4.8	Address management contract flow.	46
Figure 4.9	Server package diagram.	47
Figure 4.10	The api list of server of zk bridge system.	48

Figure 4.11	The flow of api update deposit root from Cosmos to Ethereum. . . .	49
Figure 4.12	The flow of api get claim transaction proof.	49
Figure 4.13	General Use Case.	50
Figure 4.14	Decomposed Use Case: Connect Wallet.	51
Figure 4.15	Decomposed Use Case: Disconnect Wallet.	51
Figure 4.16	Decomposed Use Case: Deposit ERC20 Token.	52
Figure 4.17	Decomposed Use Case: Withdraw ERC20 Token.	52
Figure 4.18	Sequence diagram Deposit ERC20 Token.	53
Figure 4.19	Sequence diagram Withdraw ERC20 Token.	53
Figure 5.1	The mock up UI of Deposit Tab	56
Figure 5.2	The database in zk brige system.	59
Figure 5.3	The UI of Deposit Tab after connecting keplr wallet and metamask wallet.	61
Figure 5.4	The UI of History Tab display deposit transaction history of keplr account and metamask account.	61
Figure 5.5	The UI of Withdraw Tab after click claim button in History Tab . .	61
Figure A.1	Receiver function flow in Cosmos bridge.	72
Figure A.2	Update deposit root function flow in Cosmos bridge.	73
Figure B.1	Update block header flow.	74
Figure B.2	Update deposit root flow.	75
Figure B.3	Cosmos block header functions.	76
Figure B.4	Cosmos validator functions.	77
Figure C.1	The package diagram of circuit system.	79
Figure C.2	The picture shows the calculating a new root when adding a new value in circuit.	81
Figure C.3	The picture shows the calculating a new root when adding a batch of new value in circuit.	82

LIST OF TABLES

Bảng 4.1	Sepecific Use case Deposit ERC20 token	54
Bảng 4.2	Specific attributes is in UC Deposit ERC 20 Token	54
Bảng 4.3	Sepecific Use case Withdraw ERC20 token using Hitory view	55
Bảng 5.1	List of libraries and tools used	59
Bảng 5.2	Statistics my bridge system	60

LIST OF ABBREVIATIONS

Abriviation	Full Expression
API	Application Programming Interface
Defi	Decentralize Finance
ECDSA	Elliptic Curve Digital Signature Algorithm
EdDSA	Edwards-curve Digital Signature Algorithm
EUD	End-User Development
HTML	HyperText Markup Language
IaaS	Infrastructure as a Service
tx	transaction
ZK	Zero-knowledge
zkBridge	A cross-chain solution use Zero-knownledge technology
ZKP	Zero-knowledge Proof

CHAPTER 1. INTRODUCTION

1.1 Motivation

Since the debut of Bitcoin, blockchains have evolved into an expansive ecosystem encompassing various applications and communities. Cryptocurrencies like Bitcoin and Ethereum have gained rapid traction, with market caps exceeding a trillion USD, attracting interest from institutional investors. Decentralized Finance (DeFi) has demonstrated the potential for blockchain to enable financial instruments that were previously impossible. Additionally, digital artists and content creators are utilizing blockchains for transparent and accountable circulation of their works.

The blockchain ecosystem has witnessed significant growth in heterogeneity. Numerous blockchains have been proposed and deployed, leveraging different techniques, security guarantees, and performance levels. As no single blockchain dominates in all aspects, it is envisioned that the future will involve a multi-chain environment where various protocols coexist. Developers and users will be able to choose the most suitable blockchain based on their preferences, cost considerations, and offered amenities.

Currently, Zero-Knowledge technology has gained prominence in blockchain ecosystems due to its ability to efficiently verify complex information through proofs. By employing this technology in cross-chain solutions (such as zkBridge), enhanced security can be achieved, while reducing on-chain verification costs. However, existing zkBridge solutions primarily focus on transferring assets between blockchains with similar consensus mechanisms. Bridging two blockchains with different consensus mechanisms, such as the Cosmos network and Ethereum network, presents significant challenges when applying Zero-Knowledge technology. Therefore, in this thesis, I will introduce a zkBridge that connects Cosmos and Ethereum, enabling users to transfer data between these chains in a convenient, safe, and decentralized manner.

1.2 Objectives and Scope of the Graduation Thesis

The primary objectives of this graduation thesis are as follows:

- Bridging Cosmos block header to Ethereum: The main goal is to develop a mechanism for generating Zero-knowledge proof to facilitate the seamless bridging of Cosmos block header data to Ethereum.
- Bridging deposit root transaction: This thesis will focus on constructing a Merkle tree to efficiently store deposit transactions, with the root of the tree stored on the Cos-

mos bridge. Subsequently, a Zero-knowledge proof will be implemented to bridge the deposit root transaction to the Ethereum bridge.

- Claim deposit transaction: The aim is to devise a proof generation process that enables users to claim their assets previously deposited on Cosmos through the implementation of Zero-knowledge proof mechanisms.

The scope of this thesis revolves around addressing the challenges and complexities associated with bridging essential data between the Cosmos and Ethereum networks. It will primarily focus on the generation of Zero-knowledge proofs to ensure secure and verifiable transfers of data. Additionally, the thesis will involve the development of specialized algorithms and circuits to effectively handle the verification process on both the Cosmos and Ethereum bridges. The implementation and testing of the proposed solutions will be conducted using selected programming languages and tools suitable for the respective platforms.

1.3 Tentative Solution

I will build a zkBridge based on the concept described in the paper[1], with some adjustments to align with my knowledge. The proposed solution consists of three steps, each utilizing Zero-Knowledge technology to enhance security and processing speed:

- Bridging the block header of Cosmos to Ethereum.
- Transferring transaction information from the bridged block header to Ethereum.
- Generating a proof for users to retrieve their assets in Ethereum after the bridge.

The project will make two main contributions:

- Building a bridge that connects the Cosmos and Ethereum blockchain networks, enabling users to conveniently, securely, and decentralizedly transfer data between these networks.
- Creating a Zero-Knowledge library to assist programmers who wish to implement Zero-Knowledge technology in the Cosmos network.

1.4 Thesis Organization

The remaining sections of this graduation project report are organized as follows:

- Chapter 2 presents the background knowledge necessary for this project, covering blockchain fundamentals, including Cosmos and Ethereum, types of signatures, hash functions, Merkle trees, and Zero-Knowledge technology.

- Chapter 3 provides an overview of the proposed solution and its functionality.
- Chapter 4 describes the detailed operational flow of the bridge through diagrams.
- Chapter 5 evaluates the performance of our zkBridge through experimentation.
- Chapter 6 highlights the contributions that I am most proud of.
- Chapter 7 concludes the thesis.

Considering the time limitations in building zkBridge, the current implementation of the bridge supports asset transfers from the Cosmos network to Ethereum but does not yet facilitate reverse transfers. Therefore, this thesis will primarily focus on presenting the capabilities and advancements of the Cosmos network within the bridge framework, with less emphasis on Ethereum.

CHAPTER 2. TECHNOLOGY BACKGROUND

2.1 Blockchain

Blockchain is a decentralized digital ledger that organizes data into blocks. Each block consists of a block header containing essential information such as the block number, previous block hash, timestamp, Merkle root, and nonce. These block headers establish a chronological order and link the blocks together. Validators play a crucial role in maintaining the integrity and security of the blockchain by verifying and validating transactions and blocks. Through consensus mechanisms like mining or staking, validators ensure compliance with the network's rules and prevent fraudulent activities.

Different blockchain networks may employ varying consensus mechanisms and leverage computational resources differently. For instance, Ethereum and Cosmos are examples of blockchain networks that currently use the Proof of Stake (PoS) mechanism. PoS selects validators based on the number of tokens they hold and are willing to stake. Validators validate and propose new blocks, with their selection determined by their stake. PoS is known for its energy efficiency and offers security through stake slashing to penalize malicious behavior. However, there are differences between blockchain networks in terms of signature types, block header creation methods, and other leveraging computational aspects.

2.1.1 Ethereum

Ethereum is a decentralized blockchain platform widely recognized for its support of smart contracts and decentralized applications (dApps). At its core, Ethereum employs the Ethereum Virtual Machine (EVM) as a runtime environment for executing smart contracts. Developers use Solidity, a high-level programming language specifically designed for Ethereum, to write and deploy smart contracts. Solidity enables developers to define the rules and behaviors of these contracts, facilitating the creation of diverse applications such as decentralized finance (DeFi) protocols, non-fungible tokens (NFTs), and decentralized exchanges (DEXs). The combination of Ethereum, the EVM, and Solidity provides a robust ecosystem for building and deploying innovative blockchain-based solutions.

a, ECDSA signature

In the context of smart contract development on the Ethereum blockchain, the Elliptic Curve Digital Signature Algorithm (**ECDSA**) holds paramount importance as a cryptographic tool within the Solidity programming language. **ECDSA** is widely utilized for generating and verifying digital signatures, ensuring data integrity, authentication, and non-

repudiation in decentralized applications. The **ECDSA** algorithm leverages the mathematical properties of elliptic curves to create secure and efficient digital signatures, providing a means for users to authenticate their transactions and messages on the Ethereum network. By employing **ECDSA** within Solidity smart contracts, developers can implement secure and tamper-proof mechanisms, safeguarding sensitive information and enabling secure interactions between parties without the need for a central authority.

b, Keccak256 hash function in Solidity

In the domain of smart contract development on the Ethereum blockchain, **keccak256** stands as a crucial cryptographic function within the Solidity programming language. Also known as **SHA-3** (Secure Hash Algorithm 3), **keccak256** is a one-way hash function that takes an input data and produces a fixed-size, 256-bit hash value. The significance of **keccak256** lies in its ability to ensure data integrity and security within smart contracts, as it is virtually impossible to reverse-engineer the original input from its hash output. This cryptographic function finds widespread use in various security-related applications, including creating digital signatures, generating unique identifiers, and validating data authenticity.

c, EncodePacked in Solidity

In the realm of smart contract development on the Ethereum blockchain, **abi.encodePacked** plays a fundamental role as a vital utility function in the Solidity programming language. This function serves as a powerful tool that enables developers to efficiently pack and concatenate multiple variables or data types into a single byte array without introducing any padding or encoding overhead. The significance of **abi.encodePacked** lies in its ability to optimize storage and minimize gas costs, making it particularly valuable in scenarios where compactness and cost-efficiency are essential considerations.

d, Upgradable contracts

Upgradable contracts represent a cutting-edge approach in smart contract development that offers significant advantages in terms of flexibility and adaptability. Unlike traditional contracts, which are immutable once deployed, upgradable contracts enable developers to modify and upgrade contract logic over time without the need for redeployment. This groundbreaking feature is achieved through the implementation of proxy patterns and separation of storage and logic. By decoupling the contract's logic from its data, upgradable contracts allow for seamless updates and bug fixes, eliminating the risks associated with redeployment.

The ability to upgrade smart contracts post-deployment opens up new possibilities for de-

developers, allowing them to adapt to changing business requirements, fix vulnerabilities, and implement new features while preserving the integrity of existing data. Moreover, upgradable contracts are particularly beneficial in scenarios where contracts are expected to have a long lifespan or need to comply with evolving regulatory frameworks.

e, Smart contract frameworks

OpenZeppelin and Chainlink stand as two prominent and influential frameworks in the realm of smart contract development on the Ethereum blockchain. OpenZeppelin provides a comprehensive library of reusable and audited smart contracts, offering developers a robust foundation to build secure and efficient decentralized applications. With OpenZeppelin, developers can leverage well-tested and community-vetted contracts for functionalities such as token standards (e.g., ERC20, ERC721), access control, and contract upgradeability. This framework streamlines the development process, allowing for faster deployment of reliable smart contracts without compromising on security.

Chainlink, on the other hand, introduces an essential integration for external data access in Solidity smart contracts. As a decentralized oracle network, Chainlink facilitates the secure and reliable connection between smart contracts and off-chain data sources, ensuring that on-chain applications can access real-world data in a trustless manner. This capability expands the use cases of smart contracts, enabling applications such as decentralized finance (DeFi), supply chain management, and insurance to operate based on real-time and accurate data.

f, Hardhat

Hardhat stands as a robust and highly regarded development environment tailored for Ethereum smart contract development. This versatile tool offers a comprehensive set of features and functionalities, making it a popular choice among developers for building, testing, and deploying smart contracts with efficiency and confidence. Hardhat's extensible plugin system and scripting API allow for seamless integration with various tools and libraries, enabling developers to customize their development workflow to suit specific project requirements. Moreover, its compatibility with TypeScript brings the advantages of strong typing and enhanced code readability to smart contract development, leading to more reliable and maintainable codebases. Throughout this thesis, we delve into the capabilities and benefits of Hardhat, exploring its role in streamlining the development process and empowering developers to create secure and sophisticated decentralized applications on the Ethereum blockchain.

g, Web3 ad Ethers

In the domain of Ethereum blockchain development, ethers and web3 stand as two fundamental libraries that play pivotal roles in facilitating seamless interactions with the Ethereum network. Ethers is a powerful and versatile JavaScript library that provides developers with a robust set of tools for interacting with Ethereum. It offers a simple and intuitive API for sending transactions, querying contract data, and managing Ethereum accounts and keys. Ethers' modular architecture and well-documented features empower developers to efficiently integrate Ethereum functionalities into their applications, ensuring a smooth and reliable user experience.

On the other hand, web3 serves as another crucial JavaScript library, acting as an interface between decentralized applications and the Ethereum blockchain. Web3 provides a standardized API that abstracts the complexities of interacting with Ethereum, making it easier for developers to build decentralized applications and interact with smart contracts. Its comprehensive functionalities enable users to perform a wide range of operations, such as sending transactions, reading data from the blockchain, and accessing accounts.

2.1.2 Cosmos

The Cosmos network is a decentralized ecosystem comprising interconnected blockchains that address scalability and interoperability challenges in the blockchain space. It provides a framework that enables seamless interaction and asset/data exchange between different blockchains. The network utilizes the Tendermint consensus algorithm, which ensures fast finality and Byzantine fault tolerance. Through the Inter-Blockchain Communication (IBC) protocol, Cosmos facilitates secure cross-chain transactions and communication. By creating an interconnected network of blockchains, Cosmos aims to foster collaboration, scalability, and innovation within the blockchain ecosystem, paving the way for a more interconnected and scalable future.

In the Cosmos network, cryptographic functions play a crucial role in ensuring the security and integrity of operations. The SHA256 and SHA512 hash functions are utilized for transaction identification and data storage, providing unique digital fingerprints. Validators rely on the Edwards-curve Digital Signature Algorithm (EDDSA) for block verification, while users employ ECDSA for signing transactions, guaranteeing authenticity and integrity. These cryptographic functions are essential for maintaining a secure and trusted environment within the Cosmos network.

a, Cosmos Block Header

The Cosmos block header is a metadata container that holds important information about a block in the Cosmos network. The Cosmos block header consists of several components that provide essential information about a block within the network. These components include:

- **Version:** Indicates the protocol version or format used for the block. It requires at least two bytes and a maximum of four bytes for encoding. However, most Cosmos networks use two bytes to encode this field.
- **Chain ID:** Represents the identifier for the specific blockchain network. For example, in the case of the Oraichain network, its chain ID is "Oraichain", which requires 9 bytes to store the characters (1 byte for length and 1 byte for prefix). Thus, it takes 11 bytes to encode the Chain ID of the Oraichain network.
- **Height:** Denotes the position of the block within the blockchain, indicating the number of preceding blocks in the chain. It always uses 9 bytes for encoding this field.
- **Time:** Records the timestamp of when the block was created, enabling chronological ordering. The timestamp includes both the seconds and nanoseconds. The seconds time is calculated from January 1st, 1970 (00:00:00), and currently requires 5 bytes to store. Approximately every 25 years, an additional byte will be needed to store seconds. The nanoseconds range from 0 to $10^9 - 1$, and the number of bytes used to store nanoseconds also ranges from 1 to 5 bytes. Including one byte for prefix and length for the timestamp, one byte for prefix and length for seconds time, and one byte for prefix and length for nanoseconds time, the total bytes used to encode the timestamp currently range from 11 to 15 bytes.
- **Last Block ID:** Contains the hash and part set header of the previous block in the blockchain, linking the blocks together. Most Cosmos block networks currently have one part, and the block hash requires 32 bytes. Therefore, this field requires 38 bytes for encoding: 1 byte for prefix, 1 byte for length, 1 byte for prefix of the total part, 1 byte for storing the number of total parts, 1 byte for prefix of the last block hash, 1 byte for length of the last block hash, and 32 bytes for the last block hash itself.
- **Last Commit Hash:** Represents the hash of the commit containing the signatures of validators for the previous block. This field requires 34 bytes for encoding: 1 byte for prefix, 1 byte for length, and 32 bytes for the last commit hash.
- **Data Hash:** Contains the hash of the block's transaction data or other auxiliary in-

formation. It can be referred to as the root of all transactions. Similar to the LastCommitHash field, it also requires 34 bytes for encoding.

- **Validators Hash:** Represents the hash of the validator set for the specific block, ensuring the integrity and consistency of the validator set. This field also requires 34 bytes for encoding.
- **Next Validators Hash:** Contains the hash of the validator set for the next block, facilitating a smooth transition between validator sets. It also requires 34 bytes for encoding.
- **Consensus Hash:** Represents the hash of the consensus parameters and data used for block validation. It also requires 34 bytes for encoding.
- **App Hash:** Contains the hash of the application state after processing transactions in the block. It also requires 34 bytes for encoding.
- **Last Results Hash:** Represents the hash of the results obtained from executing the block's transactions. It also requires 34 bytes for encoding.
- **Evidence Hash:** Contains the hash of any evidence submitted against validators for malicious behavior. It also requires 34 bytes for encoding.
- **Proposer Address:** Represents the address of the validator who proposed the current block. It also requires 34 bytes for encoding.

```

"header": {
  "version": {
    "block": "11"
  },
  "chain_id": "Oraichain",
  "height": "11855600",
  "time": "2023-05-31T23:52:38.2906715Z",
  "last_block_id": {
    "hash": "A68664FF0B2FDD68D1F7C035CA1AE89F83C52B2BCD34A62C9D1E8215857AB135",
    "parts": {
      "total": 1,
      "hash": "9AF8E63FACE2E6958C47CEE3E6F129E006D56B32A5AF84D9A8FACA8AC75F4FD"
    }
  },
  "last_commit_hash": "562CE3228BD5A344083028EC6F3A133E521E1D415E74D0895D451204F56D1477B",
  "data_hash": "8FB187747F88927641338CE8A353317E4A46EC2A9469A4C41C8C69F9BE45C71",
  "validators_hash": "75AEC481741F6705ABF71FD89D76791D352B63D58373B86E79FA413A7698F7BA7",
  "next_validators_hash": "75AEC481741F6705ABF71FD89D76791D352B63D58373B86E79FA413A7698F7BA7",
  "consensus_hash": "0480918C7DDC283F77BFBF91D73C44DA58C3DF8A9C8C867405D8B7F3DAADA22F",
  "app_hash": "9A36A8F0EED1871D743615C4C9AC041ED51F653942397A3852B4B2FDC76FDFC7",
  "last_results_hash": "9D2198290AF8AD25D7061241AA5881FBDCC737DA84ECC64CDB965CC44A897B95",
  "evidence_hash": "E380C44298FC1C149AFBF4C8996FB92427AE41E4649B934CA495991B7852B855",
  "proposer_address": "87948C99D5E7B73CCA094893BE4E0E9F3C7508C9"
},

```

Figure 2.1: This is an illustration of the block header for block 11855600 in the Oraichain network, which is a network built using the Cosmos SDK

In the Cosmos network, each field of the block header undergoes an encoding process to ensure proper formatting before inclusion as a leaf node in an AVL+ tree. This encoding step prepares the data within each field for integration into the tree structure. Subsequently, the SHA256 hash function is applied to calculate the root of the tree by hashing the encoded leaf nodes. The resulting root, known as the block hash of the block header, serves as a unique identifier for the block, encapsulating its essential details. By employing this approach, the Cosmos network guarantees the integrity and immutability of the block header, thereby reinforcing the security and reliability of the entire blockchain.

Furthermore, the Cosmos network exhibits a faster average block time for generating block headers compared to Ethereum. The Cosmos network has an average block time of approximately 6 seconds, whereas Ethereum has an average block time of around 12 seconds. This discrepancy in block time is a distinguishing characteristic of each respective blockchain network.

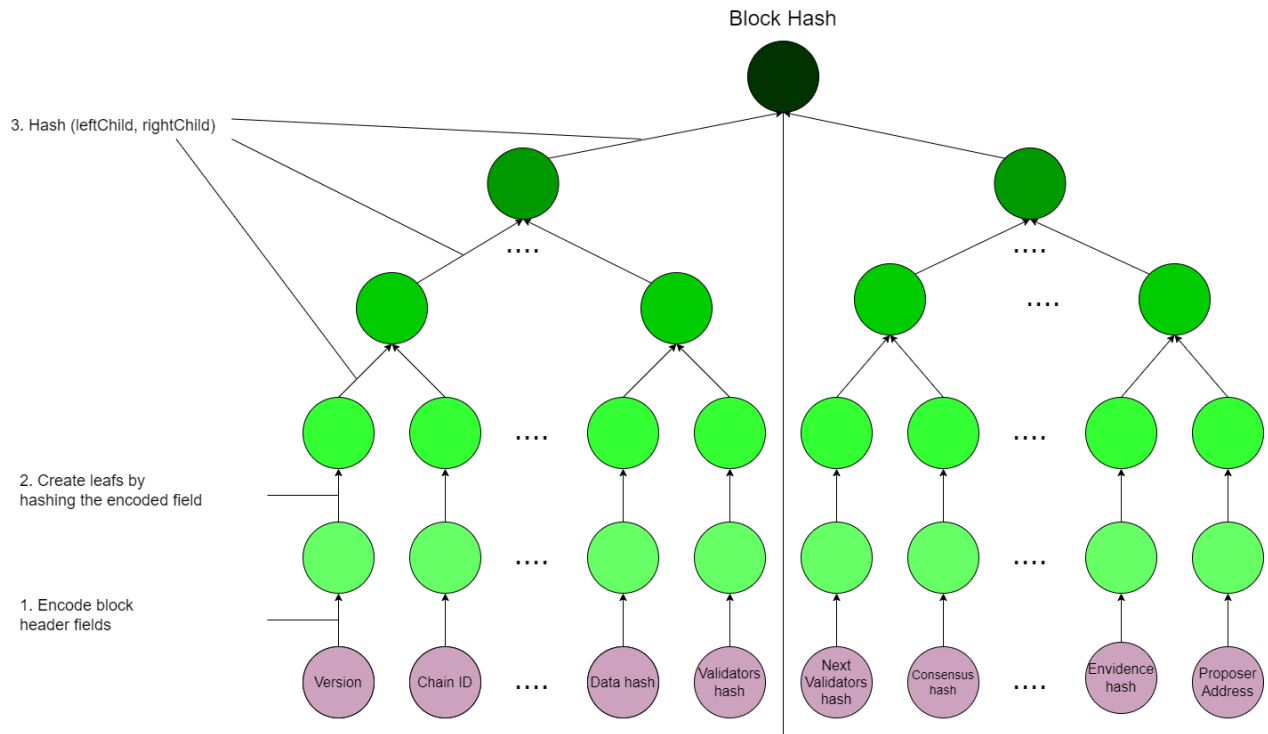


Figure 2.2: The process of creating blockHash by the fields in block header

b, Cosmos Transaction

A Cosmos transaction represents a unit of activity or operation within the Cosmos network. It enables the transfer of assets, execution of smart contracts, and other interactions within the ecosystem. The transaction data consists of three main components: **body**,

auth_info, and **signatures**.

- **Body**: This component holds the core information of the transaction, including the following fields:
 - **Messages**: Represents an array of messages within the transaction. Each message specifies a particular action to be performed. Each message includes details such as the message **type**, **sender** address, **contract** address, **msg** (specific instructions for executing the contract), and **funds** (the amount of native token used in the transaction). The byte lengths used to encode these fields are 38 bytes for the message type, 45 bytes for the sender address, and 65 bytes for the contract address. The byte length used to encode the **msg** field depends on the specific interaction with the Cosmos contract. If the **funds** field is empty (not used), it is not encoded. However, if it is not empty, the byte length used to encode this field depends on the amount of native token used and is at least 9 bytes.
 - **Memo**: Provides an optional memo or additional information associated with the transaction. This field is often empty.
 - **Timeout Height**: Specifies the timeout height for the transaction. This field is often empty.
 - **Extension Options** and **Non-Critical Extension Options**: These fields allow for potential extensions to the transaction. They are often empty.
- **Auth Info**: This component contains authentication-related information for the transaction, including the following fields:
 - **Signer Infos**: Represents an array of signer information. Each signer info includes the signer's public key, signing mode, and sequence number. The byte lengths used to encode the public key and signing mode are 72 bytes and 6 bytes, respectively. The sequence number represents the nonce of the transaction executor and requires at least 2 bytes to encode.
 - **Fee**: Specifies the transaction fee details, such as the fee amount, gas limit, payer, and granter. The byte length used to encode the fee depends on the gas limit and the specific network.
- **Signatures**: This field contains an array of signatures associated with the transaction. In this case, there is one signature represented as a base64-encoded string. The byte length used to encode the signature is 66 bytes.

```

"body": {
  "messages": [
    {
      "@type": "/cosmwasm.wasm.v1.MsgExecuteContract",
      "sender": "orai139tjpfj0h6ld3wff7v2x92ntdewungfss0ml3n",
      "contract": "orai12hzjxfh77w1572gdzct2fxv2arxcwh6gykc7qh",
      "msg": {
        "send": {
          "amount": "4360358",
          "contract": "orai1nt58gcu4e63v7k55phnr3gaym9tvk3q4apqzqccjuwppgjuyjy6sxk8yzp",
          "msg": "eyJzdWJtaXRfb3JkZXIiOnsiaYXNzZXRzIjpbeyJpbmZvIjpw7Im5hdG12ZV90b2t1biI6eyJkZW5vbSI6Im9yYWki"
        }
      },
      "funds": []
    },
    { ...
  ],
  "memo": "",
  "timeout_height": "0",
  "extension_options": [],
  "non_critical_extension_options": []
},
"auth_info": {
  "signer_infos": [
    {
      "public_key": {
        "@type": "/cosmos.crypto.secp256k1.PubKey",
        "key": "A5EAz+qeQsYC6wP4tFPuTjXJxG+aVr+fr1ZmCKG3dZ1N"
      },
      "mode_info": {
        "single": {
          "mode": "SIGN_MODE_DIRECT"
        }
      },
      "sequence": "10222"
    }
  ],
  "fee": {
    "amount": [
      {
        "denom": "orai",
        "amount": "1983"
      }
    ],
    "gas_limit": "991242",
    "payer": "",
    "granter": ""
  }
},
"signatures": [
  "iSpAjCfwdS+3/vKE4yJ6w9+VIZz8jV4CTY0o+CCJW69v9JpqJ2lwsXUhnUQT1yVpHaae86EQYzEmkCfGxJS/6Q=="
]

```

Figure 2.3: This is a representation of the transaction data within block 11855600 in the Oraichain network.

In the Cosmos network, the fields of each transaction undergo encoding and concatenation, forming the transaction data. Then, this data is then hashed using the SHA256 function, resulting in a unique transaction hash. Within a block, these transaction hashes are added as leaf nodes to an AVL+ Tree. By hashing the leaf nodes, the resulting root of the tree becomes the data hash of the block header. This data hash effectively represents the block's transactions in a condensed form, ensuring their integrity and immutability. Besides, this process establishes a secure and efficient mechanism for validating and verifying transactions within the Cosmos blockchain ecosystem, contributing to its overall security and efficiency.

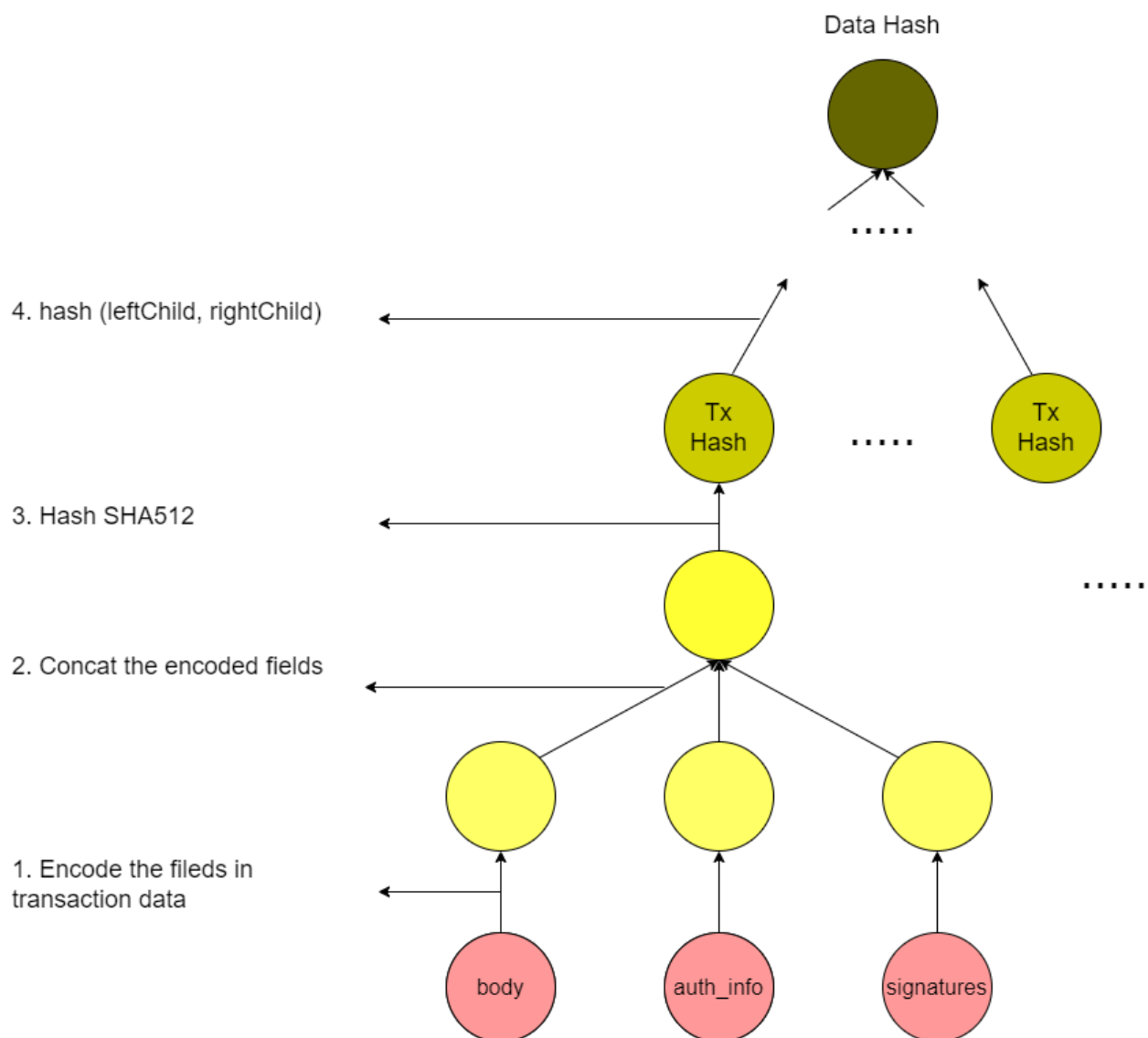


Figure 2.4: The process of calculating transactionHash and dataHash from the transactions data

c, Validator

Validators in the Cosmos network are responsible for validating transactions and proposing new blocks. They are selected based on reputation, performance, and stake in the network's native token. The validator set is a dynamic group of selected validators that participate in block validation and consensus. It is periodically updated to reflect changes in the network. Validators with higher stake have a greater chance of being selected. The validator set ensures a decentralized and secure network for efficient blockchain operations.

In the Cosmos network, validators are identified by several key fields:

- **Pubkey:** The **pubkey** field represents the public key associated with a validator. It serves as a unique identifier for the validator. The pubkey has a length of 33 bytes. Hence, the byte length used to encode the pubkey is 35 bytes (33 bytes for the pubkey + 1 byte for length + 1 byte for prefix).
- **Type:** The **type** field indicates the type of validator, reflecting the consensus algorithm or protocol they adhere to. The byte length used to encode the type is 33 bytes (31 bytes for the type value + 1 byte for length + 1 byte for prefix).
- **VotingPower:** The **votingPower** field represents the voting power of a validator. It signifies the influence or weight that a validator holds in the consensus process. The byte length used to encode the voting power depends on the specific value and encoding scheme.

```
{
  "address": "06CF27DEF0D3D7353894B01B7DCB2C1038C0F074",
  "pub_key": {
    "type": "tendermint/PubKeyEd25519",
    "value": "fwBUIPagvcOKh4d6djvaY6BnLPTjOQsfNmtJ3RGuLzI="
  },
  "voting_power": "315092",
  "proposer_priority": "-991092"
},
```

Figure 2.5: This depicts the information of a validator within the validator set of block 11855600 in the Oraichain network.

These fields collectively ensure the proper functioning and security of the Cosmos network by identifying validators, specifying their role or type, and determining their impact

on the consensus mechanism.

Validator information in the Cosmos network undergoes cryptographic encoding and hashing. First, each validator's details are encoded and hashed using the SHA256 function. Then, the resulting hashed values are then organized as leaf nodes in an AVL+ tree. After that, the root of the tree is calculated by applying SHA256 to the leaf. This root hash represents the validator hash of the block header, ensuring the integrity and efficiency of the validator information.

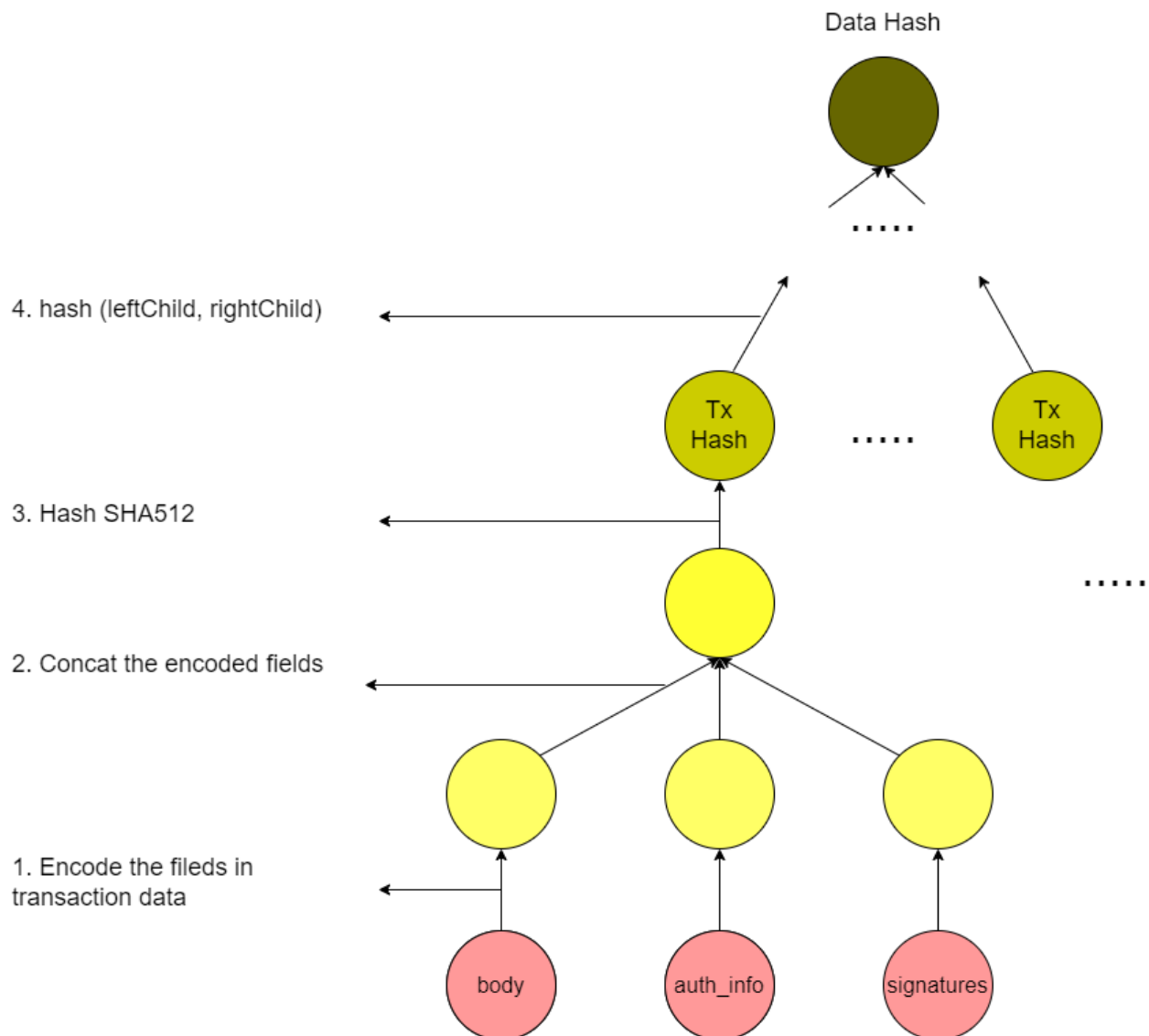


Figure 2.6: The process of calculating validatorHash from the validator set

d, Validator's signature

In the Cosmos network, validators play a crucial role in verifying and validating blocks. If a block is deemed valid by a validator, they sign it using their Ed25519 signature algo-

rithm. This signature confirms the block's authenticity and integrity, adding an extra layer of trust and security to the blockchain. To generate the validator's signature, information such as the BlockID, Height, BlockID and timestamp are encoded and hashed to create a unique message. The validator then signs this message with their private key, creating their digital signature. This process ensures that only valid blocks are accepted into the network, enhancing the overall security and reliability of the Cosmos blockchain.

```
{
  "block_id_flag": 2,
  "validator_address": "06CF27DEF0D3D735389480187DCB2C1038C0F074",
  "timestamp": "2023-05-31T23:52:44.101509099Z",
  "signature": "gVJ+UiWiCpQGuv+LLX6CzQ3dJIpNLQ9drEt1Bpto+lv7hSZ+msQYrms/uyrb58CYax1fcPvqn8BxIIJ5d2SjCA=="
},
```

Figure 2.7: This demonstrate the information of a validator's signature of block 11855600 in the Oraichain network.

e, Tendermint

Tendermint is a BFT consensus algorithm used in the Cosmos network, ensuring agreement on transaction order and validity among all network nodes. Through Practical Byzantine Fault Tolerance (PBFT), Tendermint achieves consensus when two-thirds of validators agree on the blockchain state.

An important feature of Tendermint is its support for light nodes, which enable participants to join and interact with the network without downloading and validating the entire blockchain history. Light nodes rely on Merkle proofs, cryptographic evidence of transaction inclusion or exclusion in the blockchain, to verify transaction validity and maintain a synchronized view of the blockchain state.

Tendermint's consensus process consists of proposal, pre-vote, pre-commit, and commit stages. Validators propose blocks during the pre-vote stage, and consensus is reached when two-thirds of validators pre-vote for the same block. Upon receiving enough pre-votes, the pre-commit stage begins, and when two-thirds of validators pre-commit to the same block, it becomes the committed block added to the blockchain.

To participate as a validator in Tendermint, nodes must stake a certain amount of the network's native token. Validators are selected based on stake and reputation, responsible for proposing blocks, validating transactions, and securing the network. Validators with higher stakes have a greater chance of being selected to propose blocks and participate in consensus.

By combining Tendermint's consensus algorithm with light nodes, the Cosmos network achieves a secure and scalable infrastructure. Validators ensure network integrity, while light nodes provide a lightweight way for participants to interact with the blockchain. This design empowers the Cosmos network to support various applications and use cases, establishing it as a versatile and accessible blockchain platform.

f, Cosmos SDK

The Cosmos SDK offers a modular framework for blockchain development, enabling developers to create customized modules tailored to their needs. It emphasizes interoperability, allowing communication between different blockchains within the Cosmos network. Built on the Tendermint Consensus algorithm, it provides secure and fast transaction processing. The SDK provides developer-friendly tools and libraries for application development. It incorporates governance mechanisms for stakeholder participation and supports seamless upgrades. Security and scalability are prioritized to ensure a robust and scalable network. Overall, the Cosmos SDK enables the creation of secure, interoperable, and scalable blockchains.

2.1.3 Cosmwasm

CosmWasm stands as a groundbreaking technology enabling the execution of secure and efficient smart contracts within the Cosmos blockchain ecosystem. Built on WebAssembly (Wasm), CosmWasm ensures the sandboxed execution of smart contracts, enhancing the network's resilience and minimizing potential vulnerabilities. To streamline the development process, the cw-template serves as a robust framework for creating and deploying smart contracts on Cosmos. Notably, a range of cw20 token templates has gained popularity within the CosmWasm community. These templates, such as cw20-base, cw20-staking, and cw20-atomic-swap, facilitate the creation of fungible tokens adhering to the CW20 standard. Leveraging these templates, developers can effortlessly establish token economies, enable seamless value transfer, and support diverse decentralized applications. With the combined power of CosmWasm, the cw-template, and popular cw20 token templates, the Cosmos ecosystem thrives, fostering innovation and empowering developers to create versatile and secure decentralized solutions.

2.2 Signature

2.2.1 ECDSA

ECDSA (Elliptic Curve Digital Signature Algorithm) is a cryptographic algorithm used for generating and verifying digital signatures. It is widely adopted in Ethereum and Cosmos

networks.

In Ethereum and Cosmos, ECDSA is used with the secp256k1 elliptic curve. It ensures transaction integrity and authenticity. Users create signatures with their private keys, and the network verifies them using ECDSA and secp256k1.

Both Ethereum and Cosmos rely on ECDSA and secp256k1 for secure and trusted blockchain operations. These components verify transaction authenticity and integrity, enhancing network security.

2.2.2 EDDSA

EdDSA (Edwards-curve Digital Signature Algorithm) and Ed25519 are cryptographic signature schemes used in the Cosmos network. Ed25519, based on the Edwards-curve, provides secure and efficient signature generation and verification.

Validators in Cosmos use Ed25519 to sign block headers, ensuring their integrity. Ed25519 signatures offer strong security, fast performance, and compact sizes, making them suitable for blockchain operations. Validators' signatures provide proof of validation and protect against tampering.

By utilizing Ed25519, Cosmos ensures the security and integrity of block headers, allowing participants to trust the network and maintain its decentralized nature.

2.3 Hash Function

In the field of cryptography and computer science, a hash function is a fundamental building block that plays a crucial role in various applications. A hash function takes an input, often referred to as the "message," and generates a fixed-size output, known as the "hash value" or "digest." The output is typically a sequence of bits or characters that represent the input data in a condensed form.

The primary purpose of a hash function is to ensure data integrity and provide a unique identifier for a given input. It achieves this by applying a deterministic mathematical algorithm that produces the same hash value for the same input consistently. Any slight change in the input data should result in a significantly different hash value.

Hash functions are extensively used in many areas, including data structures, digital signatures, password storage, and blockchain technology. In data structures, hash functions facilitate efficient data retrieval by mapping large data sets to smaller index values. Digital signatures rely on hash functions to ensure the integrity and authenticity of digital documents. Furthermore, blockchain technology heavily relies on hash functions to create se-

cure and tamper-proof data structures, such as Merkle trees and block hashes, ensuring the immutability of transaction records.

To be considered a secure hash function, it should possess several essential properties. These properties include:

- **Deterministic:** For a given input, the hash function should always produce the same output.
- **Preimage Resistance:** Given a hash value, it should be computationally infeasible to determine the original input.
- **Collision Resistance:** It should be highly improbable to find two different inputs that produce the same hash value.
- **Avalanche Effect:** A small change in the input should result in a significant change in the output, making it difficult to predict the resulting hash value.

In the context of Cosmos and Ethereum, several hash functions play a crucial role in ensuring data integrity, security, and consensus mechanisms. Commonly used hash functions in these blockchain ecosystems include SHA-256, SHA-512, Keccak256 and Poseidon hash.

SHA-256 (Secure Hash Algorithm 256-bit) and SHA-512 are part of the SHA family of hash functions. SHA-256, specifically, is extensively used in both Cosmos and Ethereum. It is a cryptographic hash function that generates a 256-bit hash value. SHA-256 is crucial for mining and block validation in Ethereum, providing data integrity and security for transactions and blocks.

Similarly, SHA-512 is a variant of the SHA family that produces a 512-bit hash value. While it is less commonly used in blockchain networks, it finds applications in areas where a longer hash is necessary, such as digital signatures and secure data storage.

Keccak256, a variant of the Keccak family of hash functions, is widely used in Ethereum. It is the primary hash function used to compute Ethereum addresses and provides secure hashing for various purposes within the Ethereum network. Keccak256 produces a 256-bit hash value and offers collision resistance and preimage resistance properties, making it suitable for cryptographic applications.

The Poseidon hash function is gaining traction in the blockchain community due to its efficient and secure properties. Poseidon is specifically designed for use in zero-knowledge proof systems and blockchain protocols. It combines various cryptographic operations, such as addition, multiplication, and exponentiation, to create a highly optimized hash function.

Poseidon is often used in applications requiring efficient and secure computation within the Cosmos and Ethereum ecosystems.

These hash functions, including Keccak256, SHA-256, SHA-512, and Poseidon, have undergone extensive analysis, standardization, and adoption within the Cosmos and Ethereum communities. They provide critical functionalities, including data integrity, consensus mechanisms, address generation, and cryptographic security.

In the context of our zkBridge, hash functions will be utilized in various components to provide data integrity, verification, and secure linkage between different blockchains.

2.3.1 SHA

SHA-256 (Secure Hash Algorithm 256-bit) and **SHA-512 (Secure Hash Algorithm 512-bit)** are cryptographic hash functions widely used in various applications, including blockchain technology.

SHA-256 is part of the SHA-2 family of hash functions and generates a fixed-size 256-bit (32-byte) hash value. It takes an input message of any length and produces a unique hash output that is highly resistant to collision attacks. SHA-256 is commonly used in blockchain networks like Bitcoin and Ethereum to ensure the integrity and immutability of data. It provides a reliable way to verify the authenticity of transactions and blocks. Here's the step-by-step guide on how to use the SHA-256 function to hash a value:

1. **Padding Bits:** Add extra bits to the message to ensure its length is exactly 64 bits (8 bytes) less than a multiple of 512. The first added bit should be one, and the rest should be filled with zeroes.
2. **Padding Length:** Add 64 bits (8 bytes) of data to make the final plaintext a multiple of 512 bits. Calculate these 64 bits by taking the modulus of the original message length without the padding.
3. **Initialize Buffers:** Initialize the internal state buffers used in the SHA-256 algorithm:
 - Eight buffers ($h[0]$ to $h[7]$) are initialized with default values.
 - Sixty-four round constants ($K[0]$ to $K[63]$) are precalculated and stored in an array.
4. **Compression Functions:** Break the padded message into multiple blocks of 512 bits (64 bytes) each. Process each block through 64 rounds of operations, with the output of each block serving as the input for the following block. During each round:
 - The value of $K[i]$ (round constant) is fetched from the array.

- The input message block is expanded into an array of 64 words ($W[0]$ to $W[63]$).
- A series of logical and arithmetic operations are performed on the current block, the internal state buffers, and the round constant and input words.
- The internal state buffers are updated with the result of the operations.

SHA-512, also part of the SHA-2 family, generates a fixed-size 512-bit (64-byte) hash value. It offers a higher level of security and larger hash output compared to SHA-256. SHA-512 is commonly used in applications that require stronger cryptographic protection, such as password storage, digital signatures, and data integrity checks. Here's the step-by-step guide on how to use the SHA-512 function to hash a value:

1. **Padding Bits:** Add extra bits to the message to ensure its length is exactly 128 bits (16 bytes) less than a multiple of 1024. The first added bit should be one, and the rest should be filled with zeroes.
2. **Padding Length:** Add 128 bits (16 bytes) of data to make the final plaintext a multiple of 1024 bits. Calculate these 128 bits by taking the modulus of the original message length without the padding.
3. **Initialize Buffers:** Initialize the internal state buffers used in the SHA-512 algorithm:
 - Eight buffers ($h[0]$ to $h[7]$) are initialized with predefined values.
 - Eighty round constants ($K[0]$ to $K[79]$) are precalculated and stored in an array.
4. **Compression Functions:** Break the padded message into multiple blocks of 1024 bits (128 bytes) each. Process each block through 80 rounds of operations, with the output of each block serving as the input for the following block. During each round:
 - The value of $K[i]$ (round constant) is fetched from the array.
 - The input message block is expanded into an array of 80 words ($W[0]$ to $W[79]$).
 - A series of logical and arithmetic operations are performed on the current block, the internal state buffers, and the round constant and input words.
 - The internal state buffers are updated with the result of the operations.

Both SHA-256 and SHA-512 employ the same underlying algorithm but differ in the length of the hash output. They are designed to be computationally efficient, providing a high level of security with relatively fast processing times. These hash functions are widely recognized and trusted in the field of cryptography and are essential components of secure communication and data integrity in various domains.

2.3.2 Keccak256

Keccak-256 is a cryptographic hash function that belongs to the Keccak family of hash functions. It takes an input message of any length and produces a fixed-size 256-bit hash value. The algorithm is designed to provide strong security properties, including collision resistance and pre-image resistance.

Keccak-256 is widely used in various cryptographic applications and protocols. It has gained significant attention and adoption in the field of blockchain technology, particularly in cryptocurrencies like Ethereum.

2.3.3 Poseidon

Poseidon hash is a cryptographic hash function designed for data integrity and security in various applications, particularly in blockchain technology. It uses an algebraic sponge construction, applying multiple rounds of permutation and mixing operations to process input data and produce a fixed-size hash value. The algorithm strikes a balance between security and efficiency, making it suitable for tasks like hashing transaction data and generating merkle tree roots. Poseidon hash is widely adopted in blockchain protocols to ensure the integrity and immutability of transaction records. Its efficient processing and resistance to cryptographic attacks make it a valuable tool in maintaining data integrity in decentralized systems.

2.4 Merkle Tree

Merkle trees, also known as hash trees, are data structures used to efficiently verify the integrity and consistency of large datasets. They are constructed by recursively hashing pairs of data blocks until a single hash, called the Merkle root, is obtained.

The tree structure is built by organizing the data blocks into leaf nodes at the bottom level of the tree. Each leaf node represents a small section of the dataset and is assigned a unique hash value based on its content. Moving up the tree, pairs of hashes are combined and hashed together to form the parent nodes. This process continues until a single hash, the Merkle root, is computed at the top of the tree.

To verify the integrity of a specific data block, you only need to provide the block's hash, along with a path of hashes from the Merkle root down to the corresponding leaf node. By hashing and comparing the provided data block's hash with the computed hash along the path, you can ensure that the block hasn't been tampered with and that it fits within the larger dataset.

The Merkle tree's structure allows for efficient verification because it reduces the number

of computations needed. Instead of comparing each data block individually, you only need to perform logarithmic operations based on the height of the tree.

Merkle trees are widely used in various applications, including blockchain systems like Cosmos, where they provide an efficient and secure way to verify the consistency and integrity of transactions and data stored in the blockchain.

2.4.1 AVL+ Tree

AVL+ Tree is an advanced variant of the AVL Tree, which is a self-balancing binary search tree. It is designed to provide efficient search, insertion, and deletion operations while maintaining a balanced tree structure.

The AVL+ Tree shares similar characteristics with the AVL Tree, where the heights of the left and right subtrees of each node are balanced to maintain a height difference of at most one. This balance ensures that the tree remains efficient for operations.

What sets the AVL+ Tree apart is its additional features to enhance performance and scalability. It introduces a few modifications to the traditional AVL Tree structure, such as using an array of pointers instead of individual pointers for child nodes and incorporating multiple keys per node.

By using an array of pointers, the AVL+ Tree reduces the number of memory allocations and improves cache efficiency, resulting in faster access times. The utilization of multiple keys per node enables efficient handling of large datasets, reducing the overall height of the tree and improving search performance.

The AVL+ Tree actively maintains its balanced structure by performing rotations and rebalancing operations when necessary. This ensures that the tree remains balanced even after insertions and deletions, maintaining efficient search and retrieval operations.

Overall, the AVL+ Tree is a powerful data structure that combines the benefits of the AVL Tree with additional optimizations for improved performance and scalability. It provides efficient search, insertion, and deletion operations, making it suitable for a wide range of applications that require fast and balanced data structures.

2.4.2 Fixex Merkle Tree

A Fixed Merkle Tree is a type of Merkle tree that has a fixed number of leaf nodes, with each leaf node representing a data element or a hash of a data element. The tree is constructed by recursively hashing pairs of leaf nodes until a single root hash is obtained, known as the Merkle root.

The Fixed Merkle Tree provides several benefits, including efficient data integrity verification and proof generation. By comparing the Merkle root with a previously calculated root hash, one can verify if any of the leaf nodes or their order have been tampered with.

Additionally, the Fixed Merkle Tree enables efficient proofs of membership and non-membership. A proof of membership involves providing a subset of hashes that allows a verifier to reconstruct the Merkle root and prove that a particular data element exists in the tree. On the other hand, a proof of non-membership demonstrates that a specific data element is not part of the tree.

The Fixed Merkle Tree is commonly used in various applications, including blockchain systems and file systems, to ensure data integrity and efficient verification. It is used effectively in the case when you want to prove a new root is correct based on the new leaf and the old root by showing the path from the new leaf location to the root when combined with the new leaf to create a new root, and when combined with old leaf (leaf before update, maybe empty leaf if leaf has never been updated before) creates old root.

In summary, the Fixed Merkle Tree is a deterministic and efficient data structure that provides integrity verification and proof generation capabilities. It is widely used in various domains to ensure data integrity, enable efficient proofs, and enhance the security of systems that rely on tamper-resistant data structures.

2.5 Zero-knowledge Technology

2.5.1 Defination

Zero-knowledge technology refers to a cryptographic method that allows a party (the prover) to prove the validity of a statement or claim to another party (the verifier) without revealing any additional information beyond the truth of the statement. A protocol using ZeroKnowledge Proof needs to satisfy the following three properties:

- **Completeness:** If an assertion is true, the prover can successfully convince the verifier of its truthfulness.
- **Soundness:** A prover is unable to convince the verifier of a false assertion.
- **Zero-knowledge:** The interaction between the prover and verifier only reveals the correctness of the assertion without disclosing any other knowledge.

To provide an example, consider the scenario of a person in a cave and another person outside the cave.

The person outside the cave wants to verify that the person inside the cave knows the path

from their position in the cave to the exit. However, the person outside has no knowledge of the cave's layout. To establish trust without revealing the actual path, a zero-knowledge proof can be employed.

The person inside the cave can propose a challenge to the person outside, asking them to select one of the two cave exits. The person inside will then navigate through the cave according to their knowledge of the path and emerge from the chosen exit.

If the person inside the cave consistently exits through the chosen door each time the challenge is repeated, the person outside will gradually become convinced that the person inside indeed knows the path to the exit. This confirmation is based on repeated successful verifications without disclosing the specific path or providing any additional information.

This example illustrates how zero-knowledge technology allows the person inside the cave to prove their knowledge of the path without revealing the path itself. The repeated successful verifications build confidence in the person outside, who can believe that the person inside possesses the necessary knowledge.

In real-world applications, zero-knowledge proofs rely on sophisticated cryptographic protocols and mathematical techniques to demonstrate knowledge or possession of information without divulging unnecessary details. This ensures privacy and security while enabling parties to establish trust and verify the authenticity of claims or statements.

2.5.2 ZK-SNARK

zk-SNARK (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) is a cryptographic construction that enables the generation and verification of zero-knowledge proofs. It possesses the following key characteristics:

- **Zero-Knowledge:** The proof does not reveal any information about the secret or intermediate steps, allowing the prover to convince the verifier of a statement's validity without disclosing sensitive data.
- **Succinctness:** The proofs generated by the prover are short in length, regardless of the complexity of the computation being proven. This ensures efficient transmission and verification, resulting in significant computational savings.
- **Non-Interactive:** SNARKs are non-interactive, requiring a single round of communication between the prover and verifier. This eliminates the need for multiple interactions, enabling quick and efficient verification.
- **Argument of Knowledge:** SNARKs provide an argument of knowledge, where the

prover demonstrates knowledge of a solution without revealing the actual solution. This convinces the verifier that the prover possesses the required knowledge to solve a problem without disclosing specifics.

2.5.3 Circom language

- Circom is a Domain-Specific Language (DSL) tailored for constructing arithmetic circuits for zero-knowledge proofs. It provides a specialized set of constructs and syntax optimized for expressing mathematical computations and constraints within a circuit. Circom has the following key characteristics:
 - Static Circuit Configuration: The circuit configuration is defined at the beginning and remains fixed throughout the circuit's execution, independent of input data.
 - Constraint System: Circom uses a constraint system to specify the desired behavior of the circuit. Constraints define relationships and conditions that must hold true within the circuit.
 - Declarative Syntax: Circom follows a declarative programming paradigm, allowing circuit designers to express constraints and computations without specifying their implementation details.
 - Arithmetic Circuit Construction: Circom focuses on constructing arithmetic circuits, incorporating mathematical operations like addition, multiplication, and exponentiation.
 - Integration with zk-SNARK Libraries: Circom seamlessly integrates with existing zk-SNARK libraries, such as snarkjs, to generate zero-knowledge proofs.
 - Zero-Knowledge Proofs: Circom circuits are commonly used for constructing zero-knowledge proofs, which verify specific properties without revealing sensitive information.

Signal Inputs: In Circom, signal inputs are used to represent the values provided as input to the circuit. These inputs can be dynamic and may vary for each computation. Signal inputs are typically defined using the signal keyword in Circom. They are used to represent variables or data that are specific to a particular computation or proof generation.

- **Private Inputs:** Private inputs in Circom are values that are kept confidential and are not publicly known. They typically include sensitive information or secret values necessary for the circuit's computation. Private inputs are often part of the witness or private data that the prover wants to keep hidden. They can be defined using the 'input'

keyword in Circom.

- **Public Inputs:** Public inputs in Circom are values that are known or shared publicly. These inputs are often used to define the information that needs to be disclosed or made available for verification purposes. Public inputs can be defined separately from the circuit's inputs and are typically specified using the 'input' keyword in Circom. They are used to verify the correctness of the circuit or validate the zero-knowledge proof.
- **Output Signal:** The output signal in Circom represents the computed result or output of the circuit. It is derived from the circuit's computations and constraints. The output signal is often used to verify the correctness of the computation or to validate the zero-knowledge proof.

2.5.4 Zero-knowledge Proof (ZKP)

Zero-Knowledge Proofs (ZKPs) provide a way to prove knowledge of certain information without revealing the underlying data. One popular implementation of ZKPs is zk-SNARKs (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge).

The generation of a Zero-Knowledge Proof using zk-SNARKs typically involves several steps, each with its own components:

- **Circuit Construction:** The computation is formulated as a Rank-1 Constraint System (R1CS). [2.5.5](#)
- **Circuit Compilation:** The R1CS is compiled into an intermediate representation compatible with the zk-SNARK library. This compilation step often involves converting the code into WebAssembly (Wasm), a low-level bytecode format that can be executed in a secure and efficient manner.
- **Witness Calculation:** The prover generates a witness, which is a set of input values that satisfy the constraints defined by the R1CS. The witness represents the private data the prover wants to keep confidential.
- **Trusted Setup:** A trusted setup ceremony is performed to generate public parameters known as ptau (public toxic waste). This setup process establishes the initial parameters for the zk-SNARK system and must be performed securely to ensure the integrity of the proof generation process. The size of ptau file depends on the number of constraints in R1CS. It is proportional to zkey file generation time and zkey file size. This is the phase that takes up the most time. It takes up a lot of computer resources when running

this phase. If the size of the circuit is large, the computer with a low amount of RAM may be stopped. However, it is only generated once, and we use it to generate proofs in the next step.

- **Key Generation:** Using the compiled circuit, witness, and ptau , the prover generates a proving key and a verification key. The proving key is used to construct the Zero-Knowledge Proof, while the verification key is shared publicly to allow verifiers to verify the proof. The size of verification key depends on the number of information you want to public after generating proof.
- **Proof Generation:** The process of generating Zero-Knowledge Proofs in the Ethereum bridge system involves the prover utilizing the compiled circuit (in Wasm format), the witness, and the proving key. This generation yields a compact proof, the size of which remains constant irrespective of the circuit's size. The Zero-Knowledge Proof serves as a cryptographic attestation to the validity of the witness without disclosing any sensitive information. The proof structure consists of three points, namely **pi_a**, **pi_b**, and **pi_c**, along with **public inputs** that are the desired outputs made publicly accessible. These three points, along with the public inputs, collectively demonstrate the satisfaction of the circuit's constraints.
- **Proof Verification:** Verifiers can independently validate the Zero-Knowledge Proof using the verification key and the public inputs provided by the prover. The verification process is efficient and requires minimal computation. Besides, verifier contract can be generated by verification key. After deploying verifier contract, we can verify proof on chain.

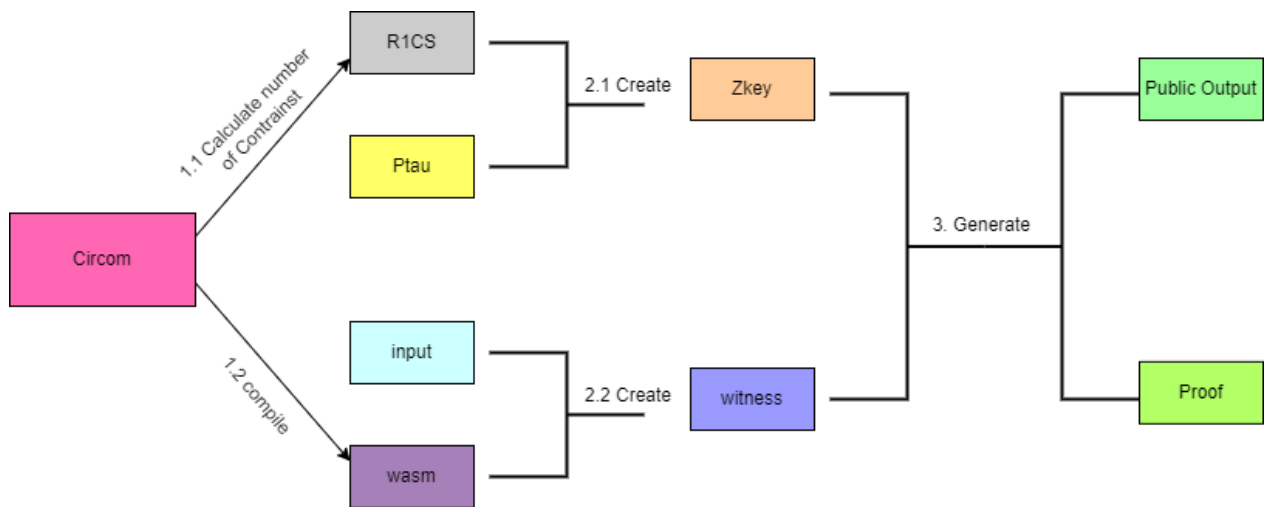


Figure 2.8: This depicts the progress generate ZK proof.

The size of a proof in Zero-Knowledge Snark (ZK-Snark) is independent of the number of constraints involved and remains constant.

2.5.5 Constraint

Constraints represents the constraints and relationships between variables. It is typically formulated as a set of equations and inequalities. The number of constraints is considered the size of circuit generating proof.

a, Signature in ZKP

The complexity of verifying a signature on a message depends on the hash function used to hash the message and the type of signature scheme applied to the hashed message.

In the context of blockchain, the complexity of verifying an ECDSA signature is considerably lower than that of an EdDSA signature. Two organizations have implemented public code for verifying ECDSA [2] and EdDSA [3] signatures in the circom language, which is commonly used in blockchain development. According to their algorithms, the verification of an ECDSA signature in circom requires 100,000 constraints, while the verification of an EdDSA signature requires 2,500,000 constraints. It is worth noting that these organizations have mentioned a method to significantly reduce the computational overhead.

In summary, the complexity of verifying a signature depends on the chosen hash function and the signature scheme employed. In the case of circom-based implementations for blockchain, the verification of an ECDSA signature has a significantly lower constraint count compared to an EdDSA signature. However, it is important to consider the specific requirements and trade-offs when selecting a signature scheme for a blockchain application.

b, Verifier contract

The Verifier contract within the Ethereum bridge system implements the verification key 2.5.4 generated by the circuit to carry out the verification of proofs. The crucial inputs of the **verifyProof** function consist of three points, namely **pi_a**, **pi_b**, and **pi_c**, along with the associated **public inputs**. Upon executing the **verifyProof** function, if it returns true, it signifies that the proof is valid, meeting all the necessary constraints imposed by the circuit. Conversely, if the function returns false, it indicates that the proof is invalid or does not satisfy the requirements. Additionally, in the **verifyProof** function, the parameters involves utilizing arrays with constant shapes, namely **pi_a**, **pi_b**, and **pi_c**. These arrays have dimensions (2,), (2, 2), and (2, 2), respectively. The specific number of input arrays utilized during the verification process is contingent upon the out value we set within the circuit.


```

    /// @return r  bool true if proof is valid
    function verifyProof(
        uint[2] memory a,
        uint[2][2] memory b,
        uint[2] memory c,
        uint[36] memory input
    ) public view returns (bool r) {

```

Figure 2.9: Example of verify proof function in contract.

c, Hash function in ZKP

The SHA256 and SHA512 hash functions are widely used in the Cosmos network. When implemented in circom, these hash functions exhibit a higher complexity, requiring approximately 50,000 and 100,000 constraints respectively to hash a 256-byte message. In contrast, the Poseidon hash function demonstrates a significantly lower constraint count, with approximately 2,000 constraints needed to hash a 256-byte message. It's important to note that while SHA256 and SHA512 are commonly utilized in the Cosmos network, the Poseidon hash function finds application in other blockchain networks and cryptographic systems.

d, Merkle Tree in ZKP

There are several types of Merkle trees that can be implemented in circom. However, since the logic of computation in circom does not depend on the input, the size of the tree needs to be configured at the beginning. The complexity of the tree construction is primarily determined by its height and the hash function used to calculate the tree root. Let's consider the example of calculating the block hash of a Cosmos block header. The block hash is computed using 14 leaf nodes, where each leaf corresponds to an attribute that is hashed using the SHA256 hash function after encoding. In the case of an AVL+ tree, it requires a total of 13 rounds of hashing to obtain the root. Therefore, when implementing an AVL+ tree in circom to calculate the block hash, it would require a minimum of $13 \times 50,000 = 650,000$ constraints in total to obtain the root.

e, Validator hash

To generate a proof for verifying the validator hash from the validator set in Cosmos, all the information of each validator in the set needs to be hashed, and then the root hash is

calculated from these hashes. In a Cosmos network with approximately 50 validators, this process requires a minimum of 99 hashing operations (50 for hashing validator information and 49 for calculating the root hash from the hashed validator set). Thus, generating a proof for verifying the Cosmos block hash involves around 5,000,000 constraints.

CHAPTER 3. METHODOLOGY

3.1 Overview

When users initiate the transfer of their assets to another chain, they perform a deposit transaction. All relevant information pertaining to the transaction is stored within the block header, specifically the block hash [b](#). By ensuring that the destination chain possesses the block hash information from the source chain, users can establish proof of their asset deposit in the source chain. This proof is subsequently utilized to facilitate asset withdrawal in the destination chain. In light of this, this thesis proposes a secure and decentralized bridge that facilitates the transfer of transaction block hashes from the Cosmos network to Ethereum. The bridge operates with a focus on preserving the integrity and security of the transferred transactions. To withdraw their assets, users are required to furnish a proof that verifies their deposit transaction in Cosmos, thereby corroborating the inclusion of their transaction within the block hash bridged to Ethereum.

3.2 Bridging Cosmos block headers

3.2.1 Bridge Cosmos block header mechanism

As discussed in section [a](#), each block header in Cosmos is signed by a set of validators, following the Tendermint consensus mechanism. When transferring a new block header from Cosmos to Ethereum, several important pieces of information within the block header need to be verified:

1. First, the signature of the Cosmos validator [d](#) on the block header is verified to ensure authenticity and integrity.
2. According to the Tendermint mechanism [e](#), the total number of validators in the new block must not exceed two-thirds of the total number of validators in the previously saved bridge block header. This constraint ensures the security and consensus of the blockchain network.
3. Additionally, the correctness of the new validator set is verified by calculating the validator hash [e](#) based on the updated set of validators. If the resulting validator hash does not match the validator hash field in the block header, the block header is considered invalid.
4. Furthermore, the block hash [a](#) is calculated from the information within the block header, including the validator hash. Comparing the calculated block hash with the

block hash signed by the validator helps confirm the validity of the block header and the integrity of the new validator set.

5. Finally, upon successful verification, the information from the new block header and the updated validator set is updated in the Ethereum network.

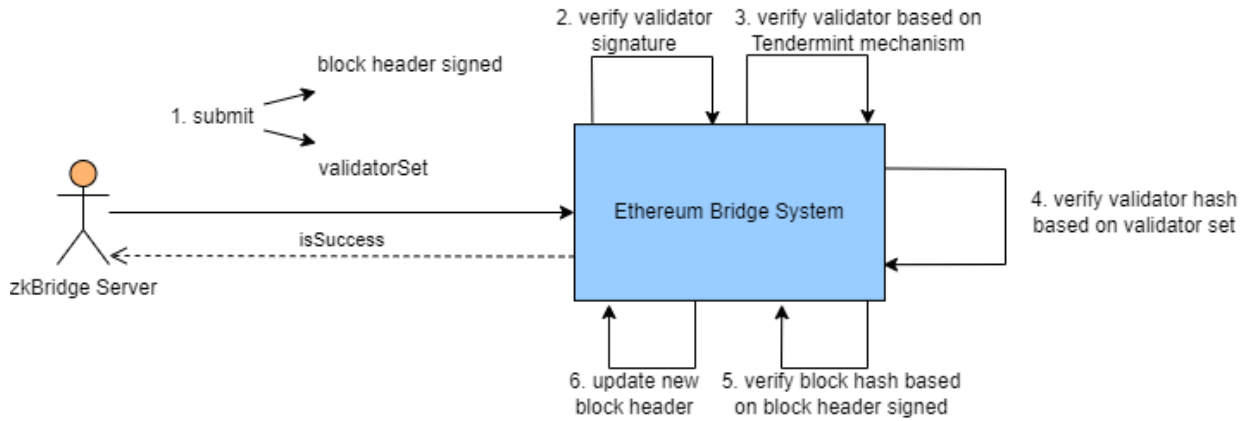


Figure 3.1: The decentralized mechanism facilitates the transfer of block headers from one chain to another

3.2.2 Evaluate the safety of the transferring Cosmos block header

Based on the constraints between the attributes used for verifying the block header, the diagram presented in Figure 3.2 illustrates their relationships.

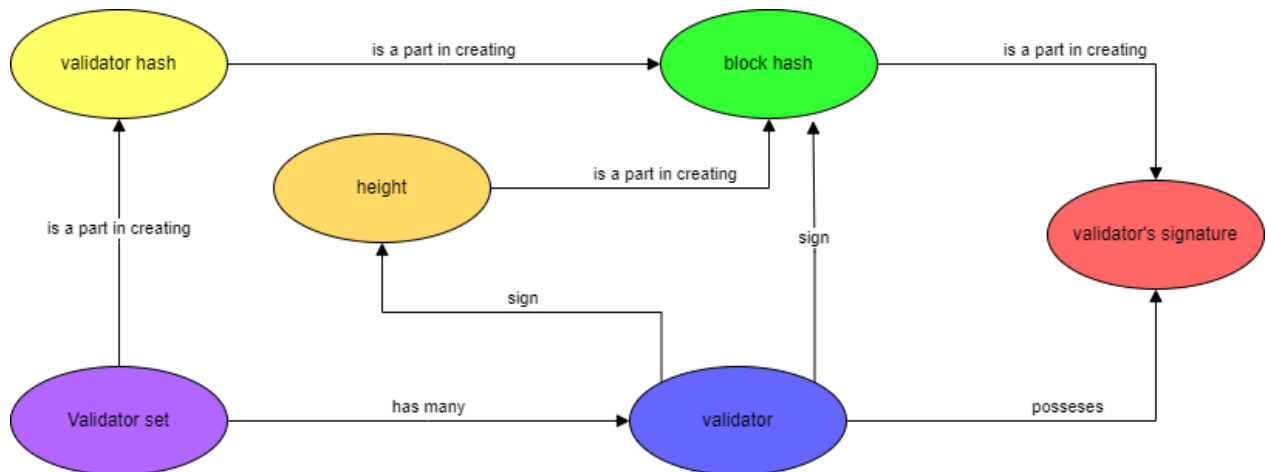


Figure 3.2: Constraints between attributes used for verifying the block header

The depicted diagram 3.2 highlights the robustness of the system against potential manipulations or errors in the block header information or the validator set. Any attempts to modify the block header data would result in incompatibility with the validators' signatures, ensuring the integrity of the block header. Similarly, any modifications to the validator set

would generate a different validator hash, which is a crucial component in the creation of the block hash. Since all validators sign the same block hash, the presence of an invalid signature associated with a block hash would lead to the failure of the update process. Moreover, even in the unlikely scenario of collusion among all validators to manipulate the block header information, it would compromise the integrity of the blockchain that relies on that specific validator set.

3.2.3 Apply Zero-knowledge technology for transferring block header

To mitigate the high gas fees associated with verifying the block header on Ethereum, I propose generating an off-chain proof for verifying the validator signature, validator hash, and block hash. This approach significantly reduces the gas fees required for on-chain verification. Additionally, the utilization of Zero-Knowledge Proofs (ZKPs) enhances the security and efficiency of the verification process. If the information within the block header is invalid, it becomes impossible to generate a valid ZK proof. Consequently, only valid ZK proofs can be generated and submitted to Ethereum for verification, ensuring the acceptance of new block headers [figure 3.3].

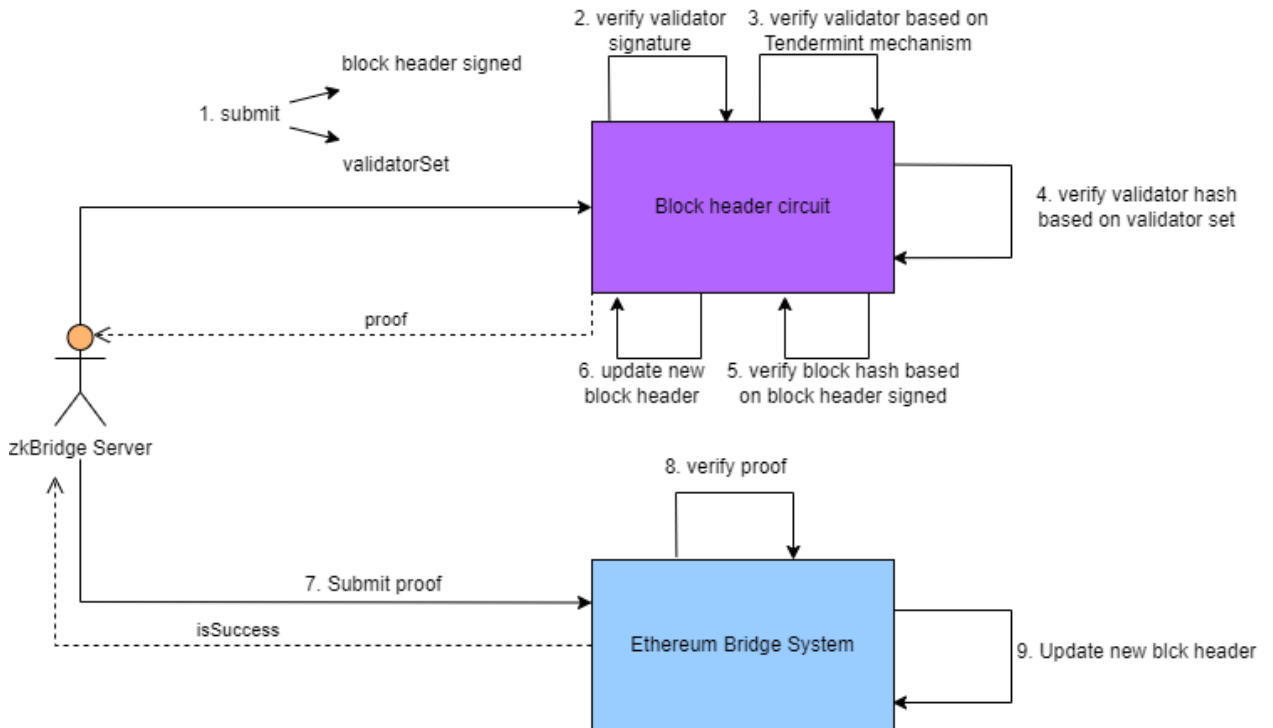


Figure 3.3: The decentralized mechanism facilitates the transfer of block headers from one chain to another with Zero-knowledge technology

3.3 Transferring deposit transactions

3.3.1 Transferring deposit transaction in a normal way

When users execute deposit transactions to transfer their assets from Cosmos to Ethereum, the information becomes part of the block hash creation process. To withdraw their assets, users need to provide additional information that demonstrates the inclusion of their transaction (Section [b](#)) in the block hash. To avoid high gas fees associated with verifying transaction data on Ethereum, users can generate an off-chain proof and verify it on-chain. The proof includes the deposit transaction data from Cosmos. If the proof is valid, the bridge contract on Ethereum will execute the corresponding action based on the proof's information. The confirmation of the block hash ensures the immutability of the included transaction data, preventing fraudulent manipulation for withdrawal purposes.

3.3.2 Transferring deposit transaction with deposit root mechanism

However, generating a ZK proof to update the new Cosmos block header to Ethereum takes several minutes. In contrast, the average time to generate each block header on Cosmos is approximately 5 seconds, significantly shorter than the time required for generating and verifying a block header's ZK proof. To address this, I propose a **deposit root** mechanism that aggregates the information of all deposit transactions within a block.

To implement this mechanism [3.4](#):

1. When a user deposits their assets to Ethereum, the deposit information is added to a queue. After a certain period of time, the queue is queried to retrieve the information.
2. Each deposit transaction in the queue is hashed to create a hash data, which is then added to a deposit tree [a](#).
3. The new root of this tree is then calculated.
4. Subsequently, a ZK proof is generated to update this new root on the Cosmos Bridge. The ZK proof requires the old root (before adding the new leaf), the new root (after adding the new leaf), the key, and the value (the new leaf).
5. Upon submitting the ZK proof to update the new root on the Cosmos Bridge, the proof is considered valid if it satisfies the logic check of the verifier contract on Cosmos. The ZK proof includes the old root of the deposit tree stored on the Bridge contract, the key, and the new value queried from the deposit queue on the Contract.

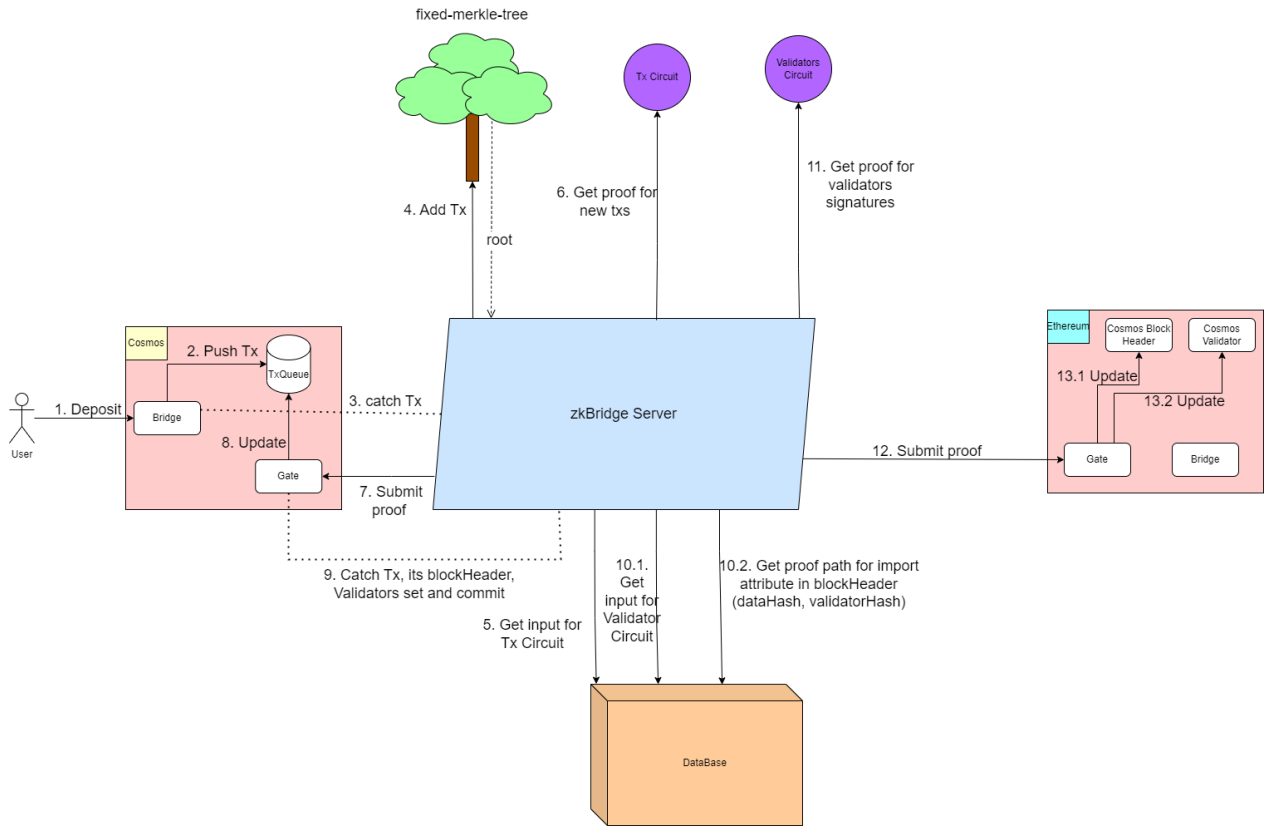


Figure 3.4: Deposit root mechanism

By utilizing data stored on Cosmos as input for the ZK proof, any attempt to use fake deposit data to deceive the system will not be compatible with the actual data on Cosmos, resulting in an invalid proof. As a result, no one can generate a fake ZK proof, while anyone can generate and update a valid ZK proof on the Cosmos bridge.

After executing the transaction to update the new root of the deposit tree on the Cosmos Bridge, the corresponding block header containing this transaction is captured and bridged to Ethereum. Following the update of the new block header on Ethereum, a proof is generated to update the new root of the deposit tree on Ethereum. Since the root of the deposit tree is a component in creating the block hash, it cannot be falsified, ensuring the integrity of the root update on Ethereum. Therefore, no one can create a fake root, while anyone can update a new valid root on Ethereum.

a, Deposit tree

In designing my deposit tree within the Ethereum bridge system, I have opted for the Fixed Merkle Tree structure 2.4.2 for its efficiency and security in managing the deposit transactions. Leveraging the compatibility of the poseidon hash function 2.3.3 with the zero-knowledge proof 2.5.4, I have chosen to utilize poseidon hash to construct both the

leaf nodes and the root of the tree. Each leaf of the deposit tree employs the poseidon hash function to securely hash essential information related to the deposit transaction, such as the destination bridge address, destination receiver address, the amount of tokens, and the token address. The key assigned to each leaf corresponds to the index of the deposit transaction within the deposit queue [Chapter 3, Transferring Deposit Transactions with Deposit Root Mechanism]. Notably, the deposit tree includes an empty leaf, which is represented as the poseidon hash of a zero element, ensuring a consistent and reliable tree structure.

b, The parameters for update deposit tree function

The verification process for updating the deposit tree in the Cosmos bridge system involves leveraging the required inputs for the **verifyProof** function within the verifier contract [b](#), utilizing the output of the **verify deposit tree root on Cosmos bridge circuit** [Chapter 4, Verify New Deposit Tree Root on Cosmos Bridge] as the public input. The inputs parameter for the update deposit tree function on the Cosmos bridge system comprises three points, namely `pi_a`, `pi_b`, and `pi_c`, which are organized into an array with eight elements, consisting of two elements of `pi_a`, four elements of `pi_b`, and two elements of `pi_c`. Additionally, the new tree root serves as a public input for verifying the proof. The Zero-Knowledge Proof also acquires the remaining public input from the Cosmos bridge. In cases where the number of deposit transactions is insufficient, the Cosmos bridge system stores a zero hash element as the value. As this proof relies on on-chain information as public input, it is imperative for the backend system to ensure precise and accurate usage of this information to generate the new deposit tree root. Any tampering or data impersonation within the backend system could result in a failed proof for updating the new deposit tree root, emphasizing the utmost importance of maintaining data integrity and security within the Cosmos bridge system.

3.4 Claiming assets on Ethereum

1. When the deposit root is updated on Ethereum, users generate a ZK proof that validates the inclusion of their deposit transaction information from Cosmos in the creation of the deposit root. [3.5](#)
2. Users submit this ZK proof to the Bridge contract on Ethereum.
3. If the ZK proof is deemed valid by the Bridge contract, it executes the corresponding action based on the proof's information and marks the proof as used.

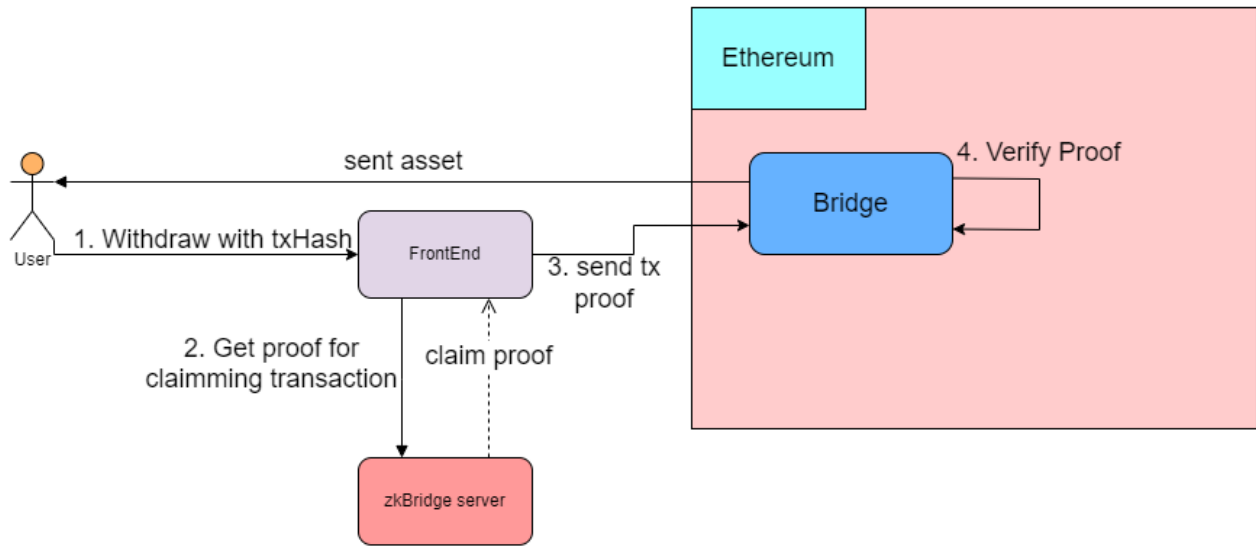


Figure 3.5: Clamming asset mechanism

This process ensures that only legitimate deposit transactions from Cosmos, backed by valid ZK proofs, are accepted and processed on the Ethereum network. By leveraging zero-knowledge proofs, the security and privacy of the deposit transaction information are preserved without revealing sensitive data.

3.5 Bridge validator set from Cosmos to Ethereum

After a considerable period with no transactions being executed, no block headers were transmitted to Ethereum, and consequently, the Ethereum Bridge’s validator set could not be updated. If a substantial change in the number of validators occurs in the new block header following an extended period, the compatibility of the tendermint mechanism integrated into the Ethereum Bridge will be compromised, resulting in the failure to update the new block header. To resolve this issue while preserving the bridge’s decentralization, a strategic approach will be employed. Specifically, the validator set will be updated periodically, taking into account the amount staked by validators, as described in [c](#). When validators stake their assets to assume the role, they will execute stake transactions. These transactions will be bridged to the Ethereum Bridge in a manner similar to the bridging process mentioned earlier, facilitating the update of the new validator set.

CHAPTER 4. SYSTEM DESIGN

4.1 Architecture Overview

Due to time limitations in the course of this thesis project, the final section titled "Bridge Validator Set from Cosmos to Ethereum" will be deferred for future research. The architectural overview, as illustrated in Figure 4.1, encompasses various components, including the application, server, circuit, Cosmos contract system, and Ethereum contract system 4.1.

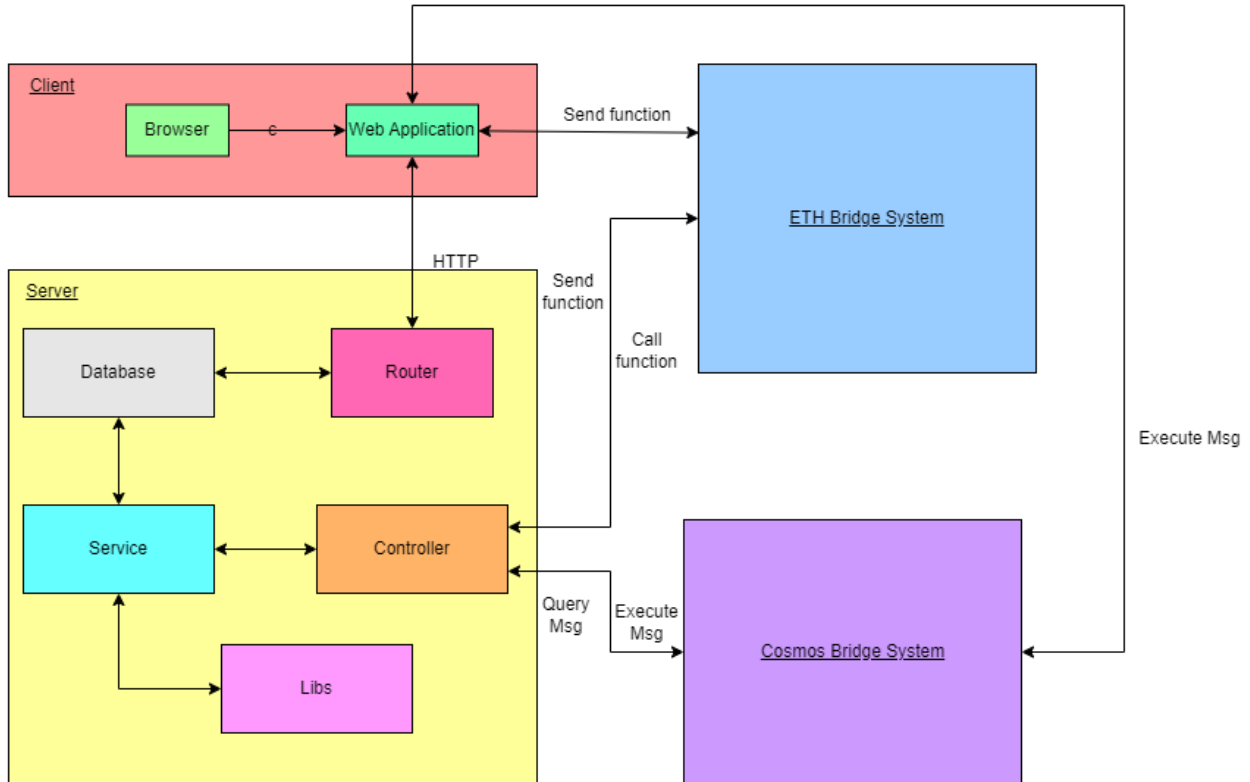


Figure 4.1: Architecture overview.

4.2 Cosmos contract system

The Cosmos contract system employed in this thesis leverages the versatile cw-template 2.1.3 to facilitate a range of essential functionalities. The system comprises eight key functions, each serving a specific purpose. The supportTokenPair function enables the addition of new token pairs to the contract, expanding the system's versatility for cross-chain asset transfers. The receive CW20 token function allows users to deposit CW20 tokens into the contract securely. Additionally, the system ensures seamless updates to the deposit tree with the update deposit tree function, maintaining data integrity and accuracy. On the withdrawal front, the withdraw CW20 token function enables users to safely retrieve their deposited to-

kens. The query deposit tree function provides users with access to essential information regarding the current state of the deposit tree. Furthermore, users can query the deposit transactions that have already been added to the tree or are still in the queue through the respective functions. Lastly, the system enables users to query the supported token pairs, ensuring transparency and accessibility to relevant information. By incorporating the cw-template and these functions, the Cosmos contract system empowers efficient and secure cross-chain interactions within the decentralized ecosystem [Figure 4.2.]

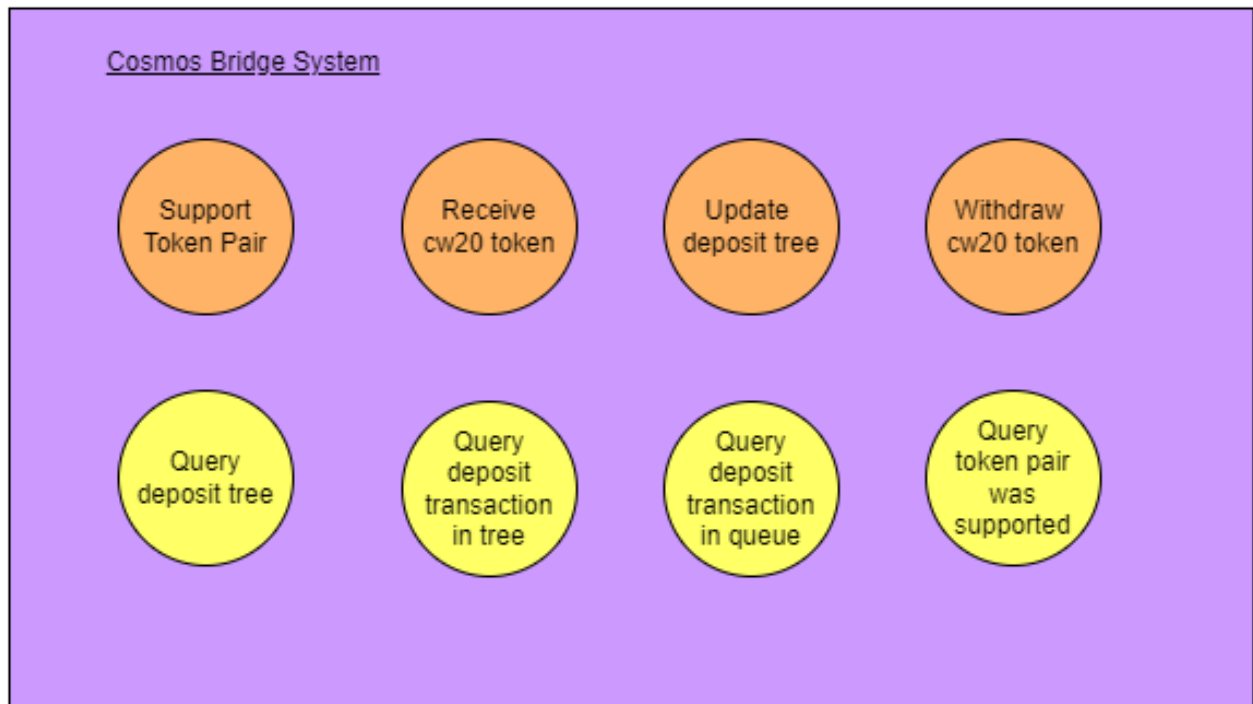


Figure 4.2: Cosmos contract system overview.

In Appendix A, I will provide a comprehensive and detailed presentation of the functions within the Cosmos bridge system [Appendix A A]. This section will serve as a comprehensive reference, offering in-depth insights into the functionalities, implementation, and usage of each function. The presentation will be structured and organized to facilitate ease of understanding and navigation for readers seeking to gain a thorough understanding of the Cosmos bridge system and its various operations. The information presented in Appendix A will complement the main body of the thesis, providing readers with additional context and technical details to enhance their comprehension of the Cosmos bridge system's inner workings.

4.3 Ethereum bridge system

The Ethereum bridge system, integrated with the powerful OpenZeppelin framework [e](#), prioritizes user privacy and user-friendliness. This bridge system encompasses four main types of contracts, each serving a distinct and critical function. The first type, updater contracts, assumes the responsibility of updating new information from the Cosmos blockchain into the Ethereum system. This includes updating Cosmos block headers, Cosmos validators, and deposit roots. Secondly, bridge contracts manage the deposit tree within the Ethereum system and facilitate smooth asset transfers. Users can deposit assets to Cosmos or claim assets previously deposited on the Cosmos bridge through this contract. The third type, verifier contracts, plays a pivotal role in verifying new information originating from Cosmos before updating the Ethereum system and ensuring a secure asset claiming process. Lastly, contract address management contracts oversee the management of all contract addresses within the Ethereum bridge system. The seamless cooperation of these four types of contracts guarantees the active and efficient functioning of the Ethereum bridge system, ensuring a smooth cross-chain experience for users [Figure 4.3].

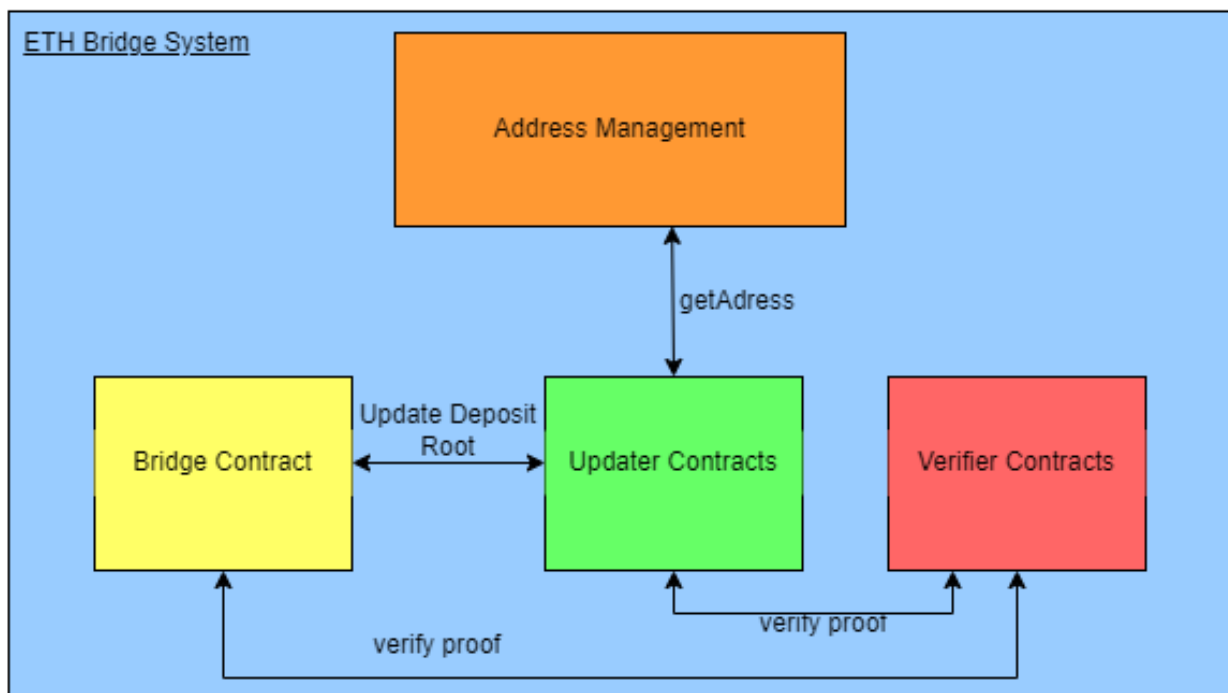


Figure 4.3: Ethereum bridge system overview.

4.3.1 Updater contracts

As previously mentioned, updater contracts play a vital role in updating new information from Cosmos to Ethereum within the bridge system. This information includes Cosmos

block headers, validator sets, and deposit roots. The updater contracts consist of three individual contracts: the gate contract, Cosmos block header contract, and Cosmos validator contract. The gate contract serves as the entry point for receiving new information from Cosmos. Before storing this information, the gate contract utilizes verifier contracts to ensure its accuracy and validity. Subsequently, the Cosmos block header contract stores the received information, encompassing block hash, data hash, and validator set details. These stored details will be utilized in the subsequent updating process 3.2. Finally, the Cosmos validator contract stores pertinent information about validators, which plays a key role in verifying the presence of the new validator set within the new block header. By meticulously managing these contracts, the updater contracts efficiently synchronize and update essential information between the Cosmos and Ethereum blockchains, contributing to the seamless functioning of the bridge system [Figure 4.4].

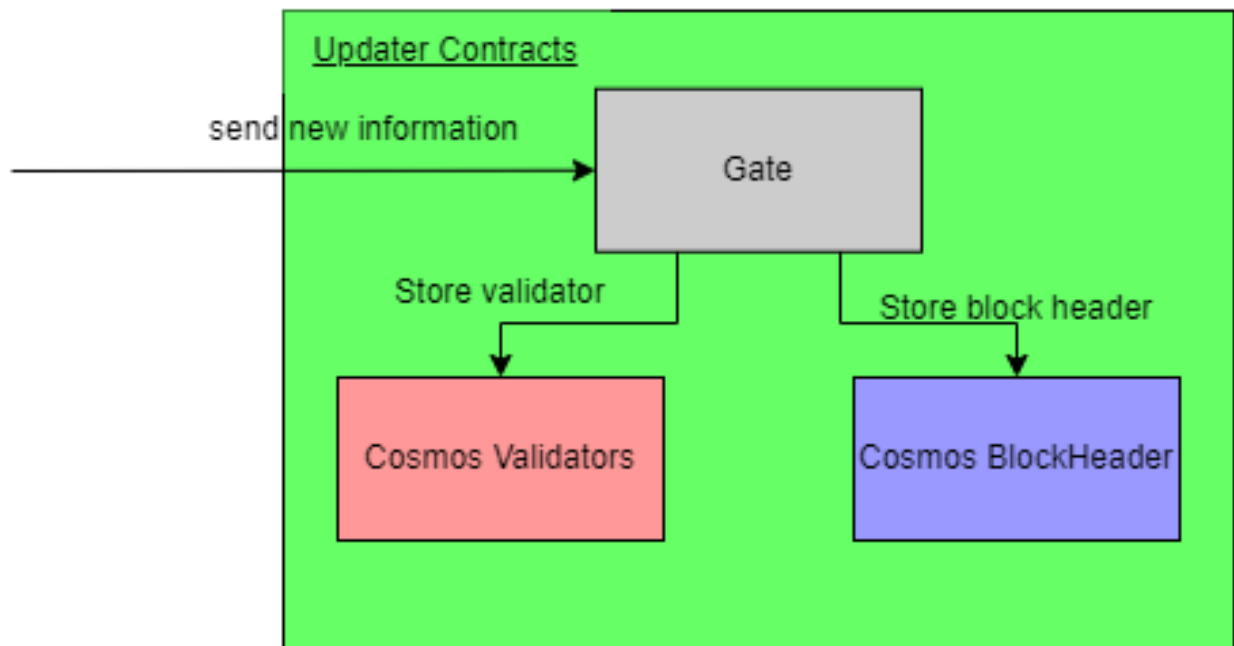


Figure 4.4: Updater contracts overview.

In Appendix B, a comprehensive presentation of the Updater contracts will be provided, offering a detailed exploration of each contract’s functionalities and operations [Appendix B]. The section will delve into the specific roles and responsibilities of each Updater contract, elucidating the mechanisms they employ to update new information from the Cosmos blockchain into the Ethereum bridge system.

4.3.2 Bridge contract

The Bridge contract within the Ethereum bridge system plays a pivotal role in managing the deposit tree and facilitating user asset claims for assets previously deposited on Cosmos. To ensure the accuracy of asset claims, the contract stores the last 30 deposit roots, thereby ensuring that users can utilize older proofs containing previous roots even after the bridge system updates with new deposit roots. When users wish to claim their deposited assets based on a specific deposit root, they must submit a proof validating their deposit transaction's inclusion in that particular deposit root. The proof is then verified by the proof contract, and upon successful validation, the Bridge contract proceeds to mint the corresponding asset for the user. To enhance security, the information in the proof is encoded **c** and hashed using the keccak256 function **b** before marking the status of this proof as true. Additionally, as the information is hashed with the key field of each deposit transaction, the claim proof for this deposit transaction can only be used once. This comprehensive process ensures the security and integrity of asset claims, enabling users to seamlessly retrieve their assets within the Ethereum bridge system [Figure 4.5].

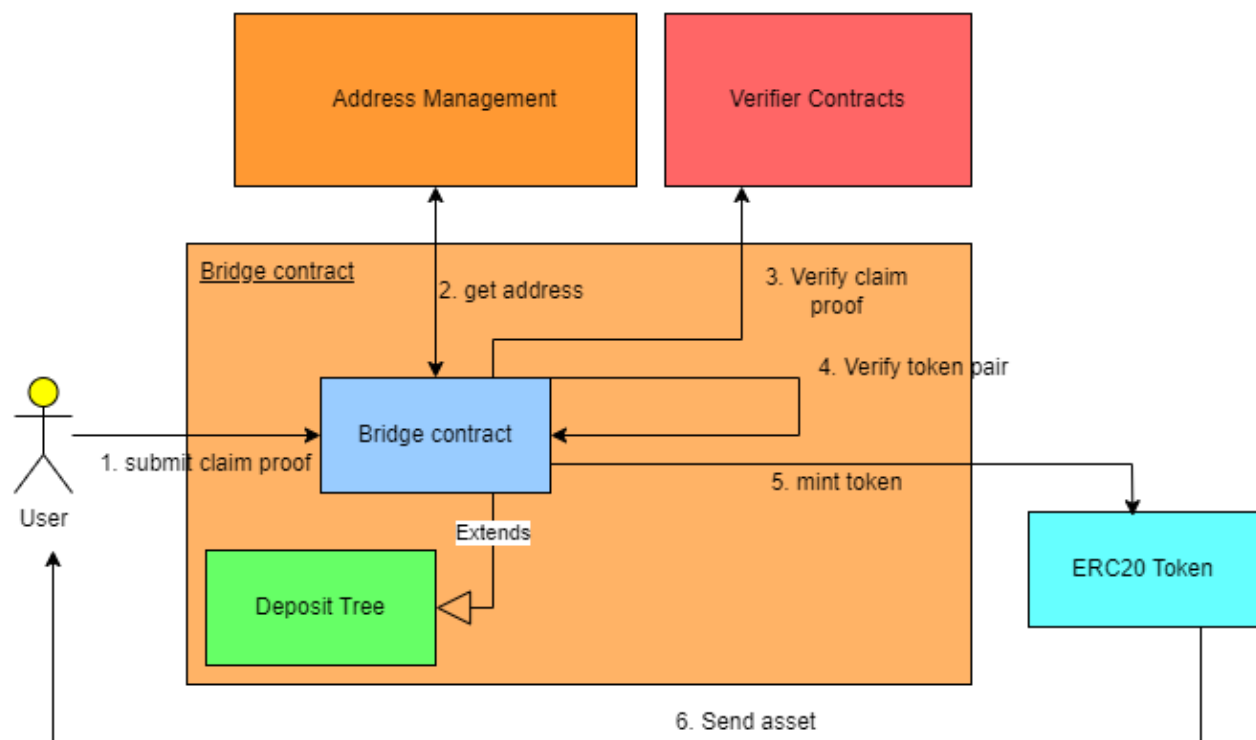


Figure 4.5: Claim transaction flow.

Furthermore, to promote interoperability and accommodate multiple bridges on the Cosmos network seeking to bridge their assets to Ethereum, the system offers a registration

mechanism through the **registerBridge** function. To qualify for inclusion in the ZK Cosmos bridge system, the bridges must meet specific criteria. Additionally, for new token pairs seeking participation in the system, they are also required to register via the **registerTokenPair** function. These registration functions can only be executed by the bridge admin, who assumes the responsibility of verifying the legitimacy and compatibility of the token pairs and Cosmos bridges. The bridge system extends its functionalities to provide essential services, including verifying the support for Cosmos bridges, checking the validity of claimed proofs, retrieving the Cosmos token address based on the Ethereum token address, and vice versa. By offering these features, the bridge system ensures seamless cross-chain interactions and reinforces its commitment to facilitating efficient and secure asset transfers between Cosmos and Ethereum networks [Figure 4.6].

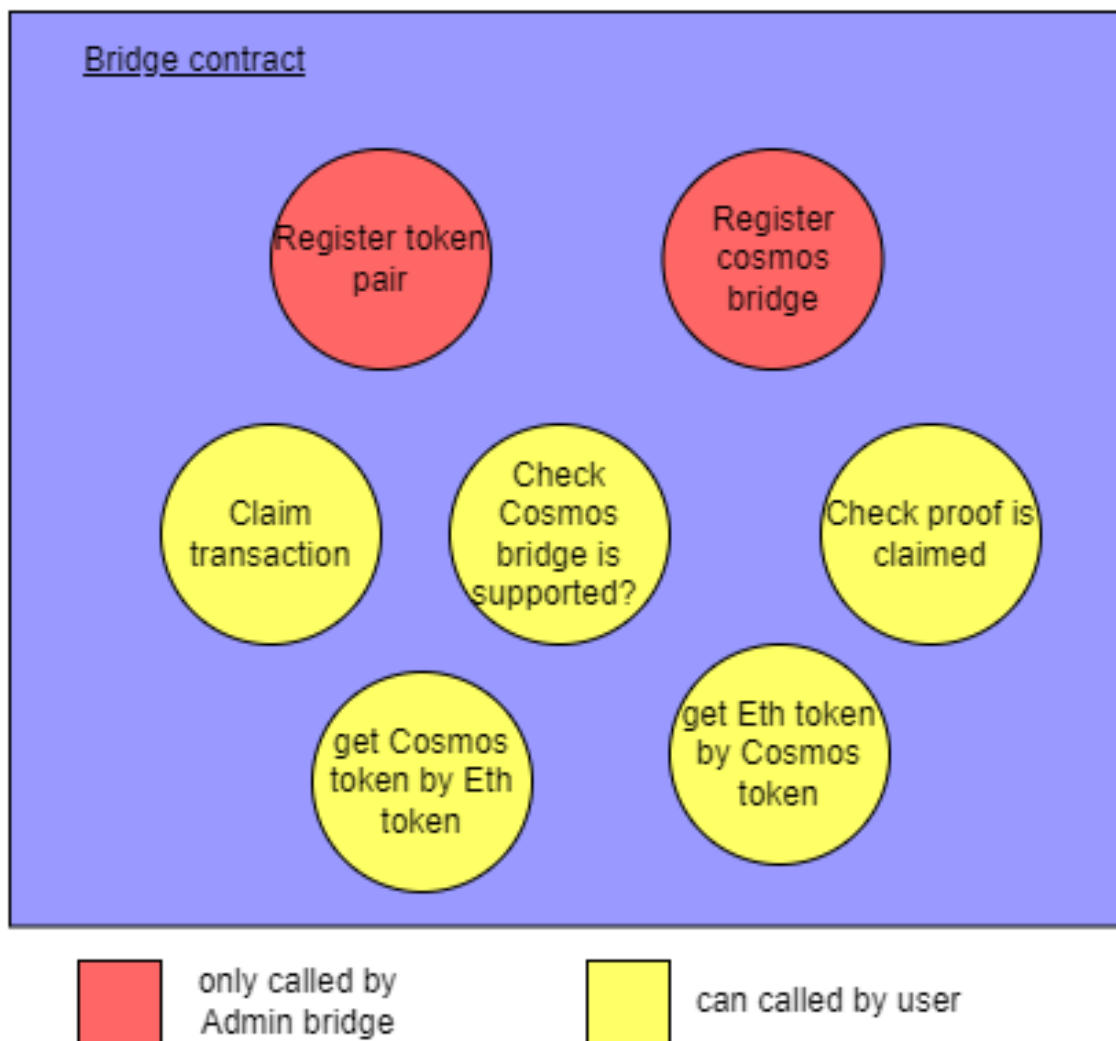


Figure 4.6: Bridge contract function.

4.3.3 Verifier contracts

The verifier contracts within the Ethereum bridge system are dynamically generated using the ZK-Snark library 2.5.2 based on their corresponding circuits 2.5.3. These contracts hold a pivotal role in the system's integrity and security by carrying out essential verification processes. They are specifically designed to validate various critical aspects, including the authenticity of validator signatures, the validity of deposit roots in deposit transactions, the accuracy of block hashes in block headers, and the presence of user's deposit transactions within the deposit tree. Leveraging the capabilities of the ZK-Snark library, these contracts ensure a robust and efficient verification mechanism, contributing to the trustworthiness and seamless operation of the Ethereum bridge system [Figure 4.7].

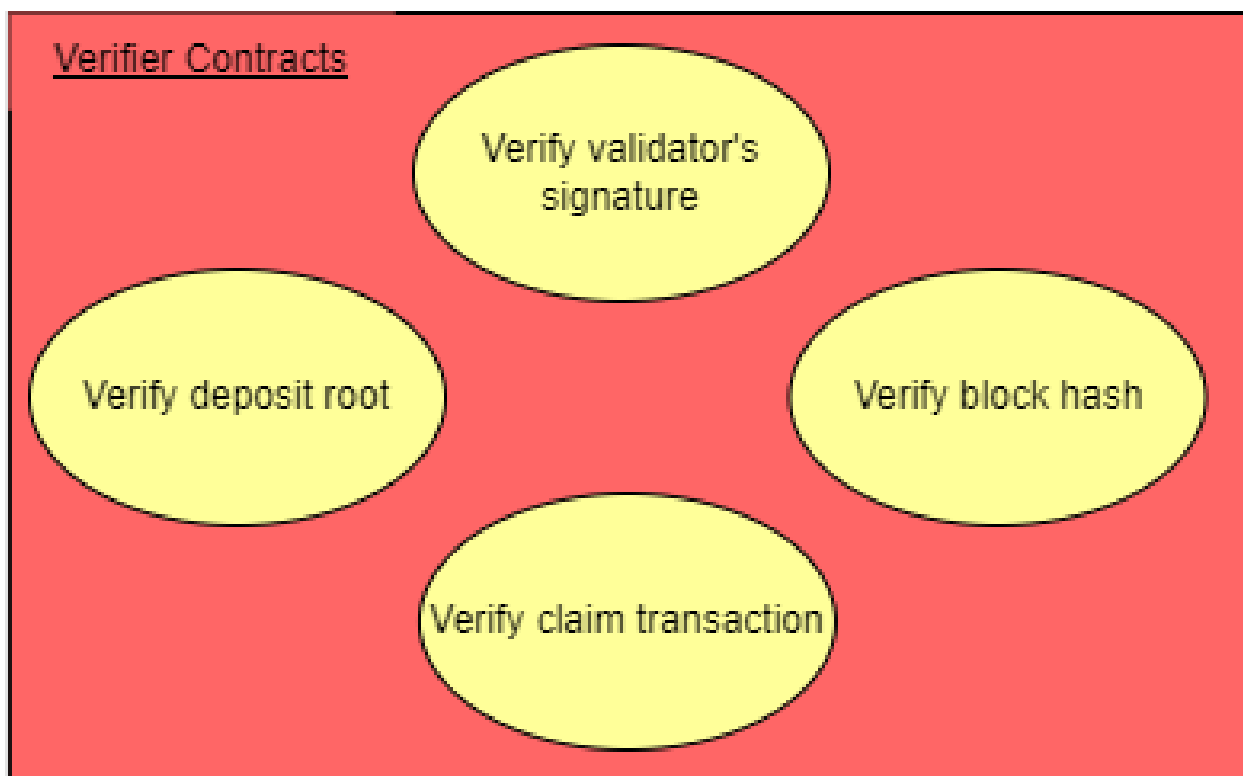


Figure 4.7: Verifier contracts function.

4.3.4 Address management contract

The Address management contract serves a crucial role within the Ethereum bridge system by storing the addresses of all contracts used in the system. During the deployment or testing phase, whenever a new contract is deployed or an existing contract's address is fixed, its address is set in this contract management. This is achieved through a mapping mechanism that associates the contract's name with its address. The name of the contract

is hashed using the keccak256 function [b](#) and then mapped one-to-one with its corresponding address. This approach allows other contracts within the system to obtain the address of a specific contract by providing its name as a parameter. As a result, if Contract B is redeployed during the testing phase, there is no need to initialize its address in the contracts that interact with it. Additionally, if a new contract address is deployed, other contracts can easily retrieve its address from the address management contract by transferring its name as a parameter in a function call. This streamlined process reduces the size of contracts and ensures a smooth and efficient interaction between them. Importantly, only the bridge address has the authority to set the addresses of contracts. If a user transfers a name of a non-existent contract to the address management contract, it will return an invalid address (address zero), and the transaction will fail, thereby preserving the security and integrity of the system [Figure 4.8].

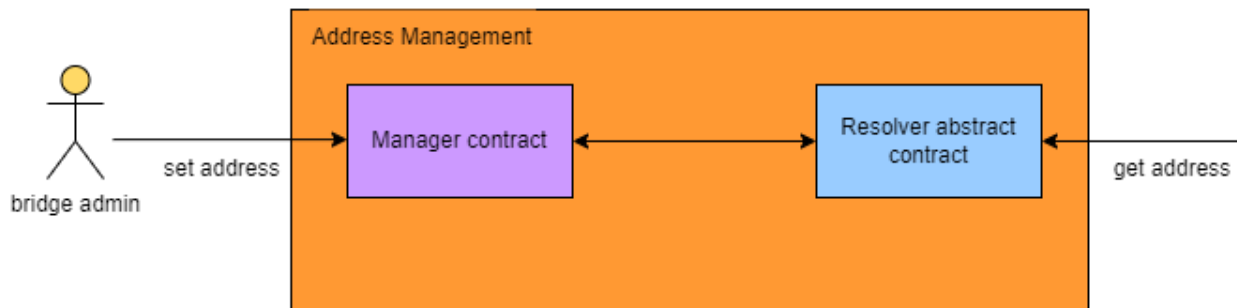


Figure 4.8: Address management contract flow.

4.4 Server

The server architecture of my system is designed to efficiently handle various functionalities through a well-structured approach, encompassing several key components. These components include the Router, Controller, Service, Model, and Libs package. The Router serves as the entry point for incoming requests, directing them to the appropriate Controller based on their endpoint. The Controller acts as the coordinator, processing the requests, and orchestrating the flow of data between the Service and the Model. The Service layer acts as an intermediary, containing essential business logic and handling data processing operations. It interacts seamlessly with the Libs package, which includes specialized packages for circom, Go, Rust, and Python. These libraries are critical in enabling the Service layer to perform advanced operations and interact with various external functionalities. The Model represents the data and database management layer, responsible for data storage, retrieval, and processing. Together, these components create a modular and maintainable server architecture, ensuring efficient communication and interaction within the system. The utilization

of specialized libraries within the Libs package enhances the system's capabilities, enabling it to perform complex tasks with ease. This well-defined architecture is crucial in facilitating scalability, flexibility, and performance within my server system, making it highly suitable for handling diverse functionalities and meeting the demands of my application.

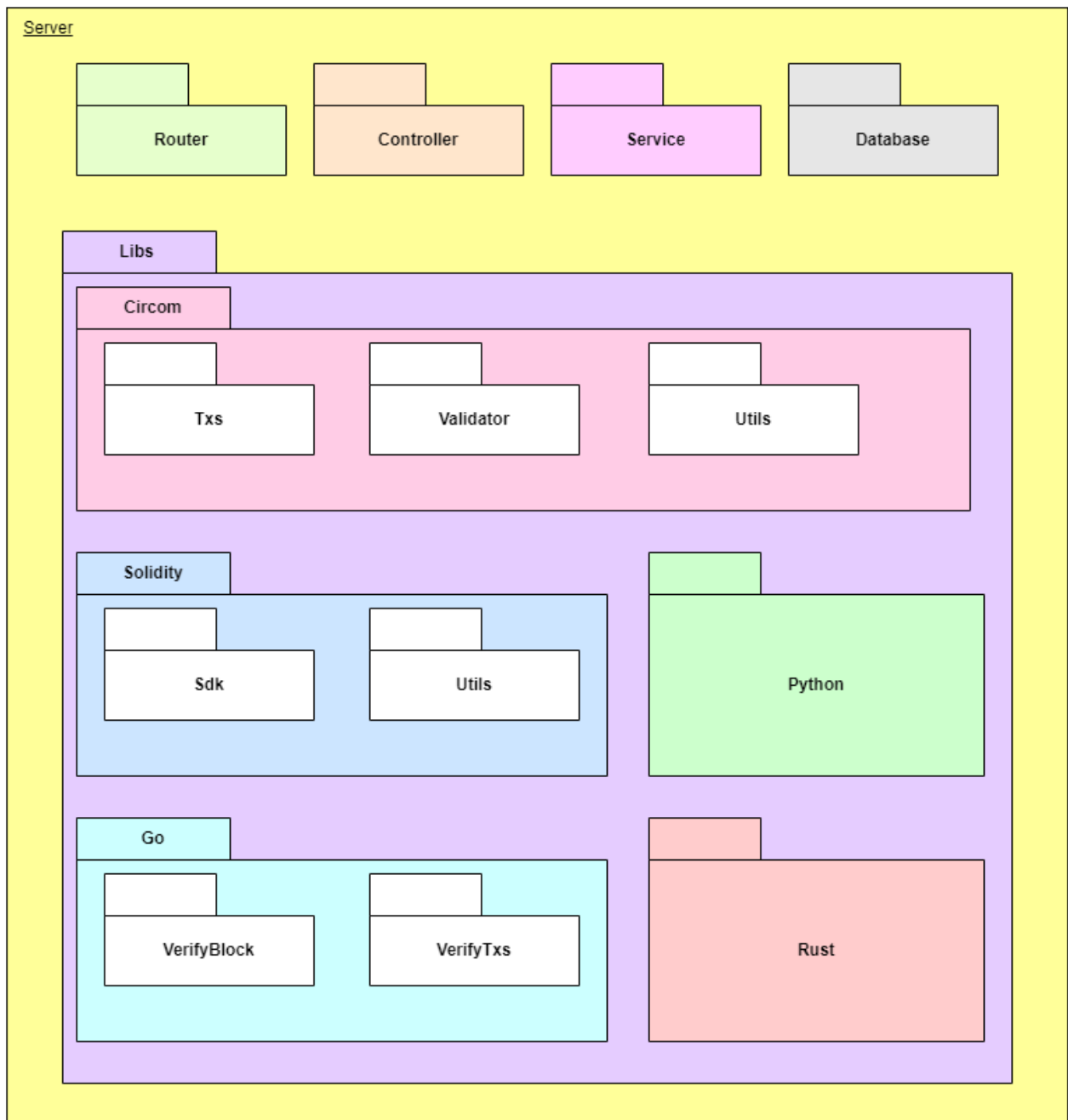


Figure 4.9: Server package diagram.

4.4.1 Zk bridge system api

The server architecture of my system comprises a total of 8 APIs, each serving distinct functions to facilitate seamless interactions within the application. Among these APIs, 5 are dedicated to handling deposit root transactions from Cosmos to Ethereum, ensuring secure and efficient asset transfers between the two blockchains. The remaining 3 APIs are specifically designed to aid the bridge admin in managing the token pairs participating in the Zero-Knowledge (zk) bridge system. Notably, two APIs hold significant importance in the system's overall functionality. The first API is responsible for updating deposit transactions to the Ethereum bridge, playing a crucial role in ensuring the accurate and timely transfer of assets. The second vital API is responsible for retrieving the necessary proof for claim transactions. These two APIs serve as the backbone of the bridge system, ensuring smooth and secure asset transfers [4.10](#).

default			^
GET	/api/query	Query	▼
DELETE	/api/db	Delete db	▼
PUT	/api/update	Gen proof	▼
GET	/api/proof/{key}	Gen proof	▼
GET	/api/infor	Gen proof	▼
token-pair			^
POST	/api/token-pair	Add token pair	▼
DELETE	/api/token-pair	Add token pair	▼
GET	/api/token-pair	Add token pair	▼

Figure 4.10: The api list of server of zk bridge system.

a, Update deposit root transaction api

This API serves a crucial function in enabling users to publish the deposit tree root of transactions on the Cosmos bridge to the Ethereum blockchain. The flow of this API is depicted in [Figure 4.11](#), showcasing the step-by-step process of updating the deposit root from Cosmos to Ethereum. When the Controller initiates the execution of this API, it triggers six distinct calls to the Service layer. These actions within the Service layer play a pivotal role in processing and validating the deposit tree root data before updating it on the Ethereum blockchain. For a comprehensive understanding of each action and its significance, detailed descriptions are provided in [Appendix D D](#). This API's seamless and secure functionality ensures the efficient transfer of deposit tree root data, enabling users to interact effortlessly

between the Cosmos and Ethereum blockchains while maintaining data integrity and security.

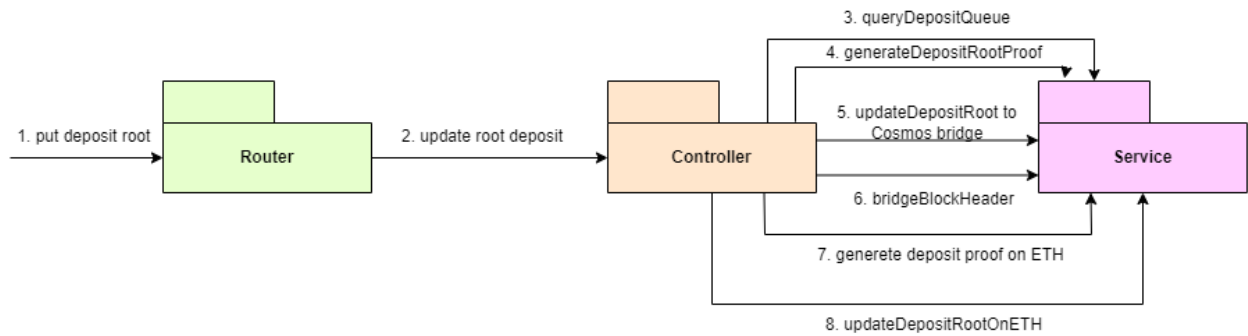


Figure 4.11: The flow of api update deposit root from Cosmos to Ethereum.

b, Get claim transaction proof api

The flow of the "Get Claim Transaction Proof" API is illustrated in the figure below [Figure 4.12].

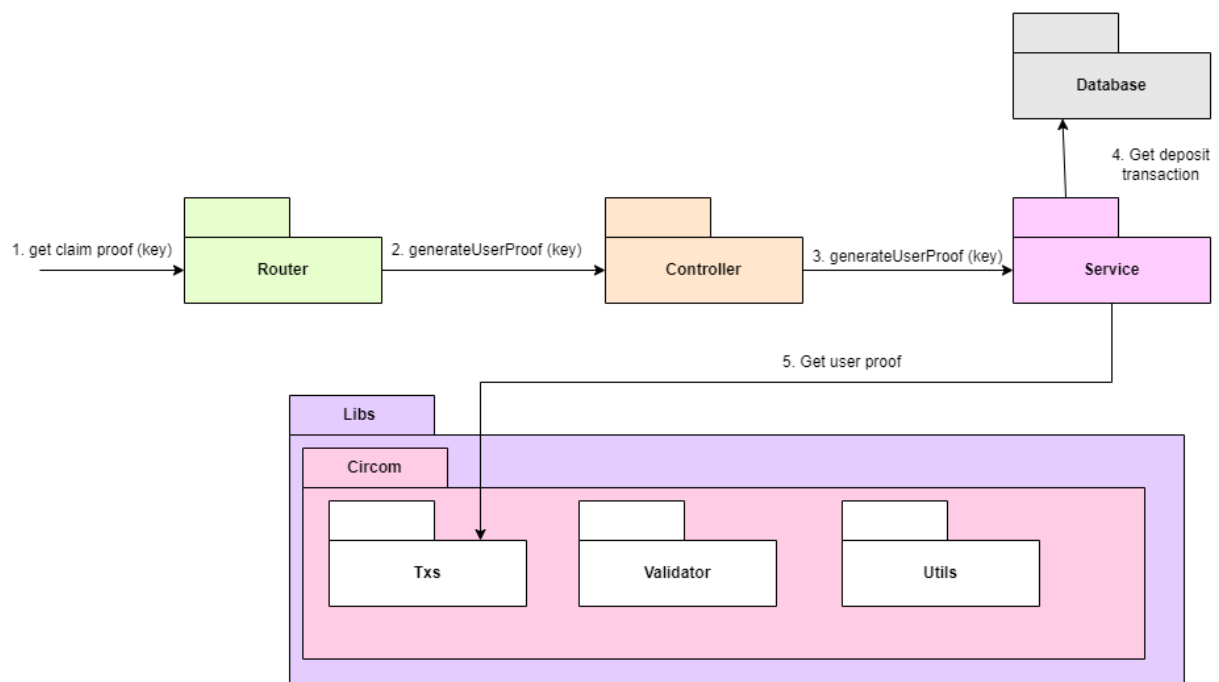


Figure 4.12: The flow of api get claim transaction proof.

4.5 Client

4.5.1 Functional Overview

Due to time constraints during the thesis project, the last section, "Bridge Validator Set from Cosmos to Ethereum," will be postponed for future work. As of now, the Bridge ap-

plication focuses on addressing four primary use cases:

1. Deposit ERC20 tokens from Cosmos to Ethereum.
2. Withdraw ERC20 tokens on Ethereum.
3. View the bridge transaction history.
4. Obtain proof of withdraw ERC20 tokens.

a, General Use Case Diagram

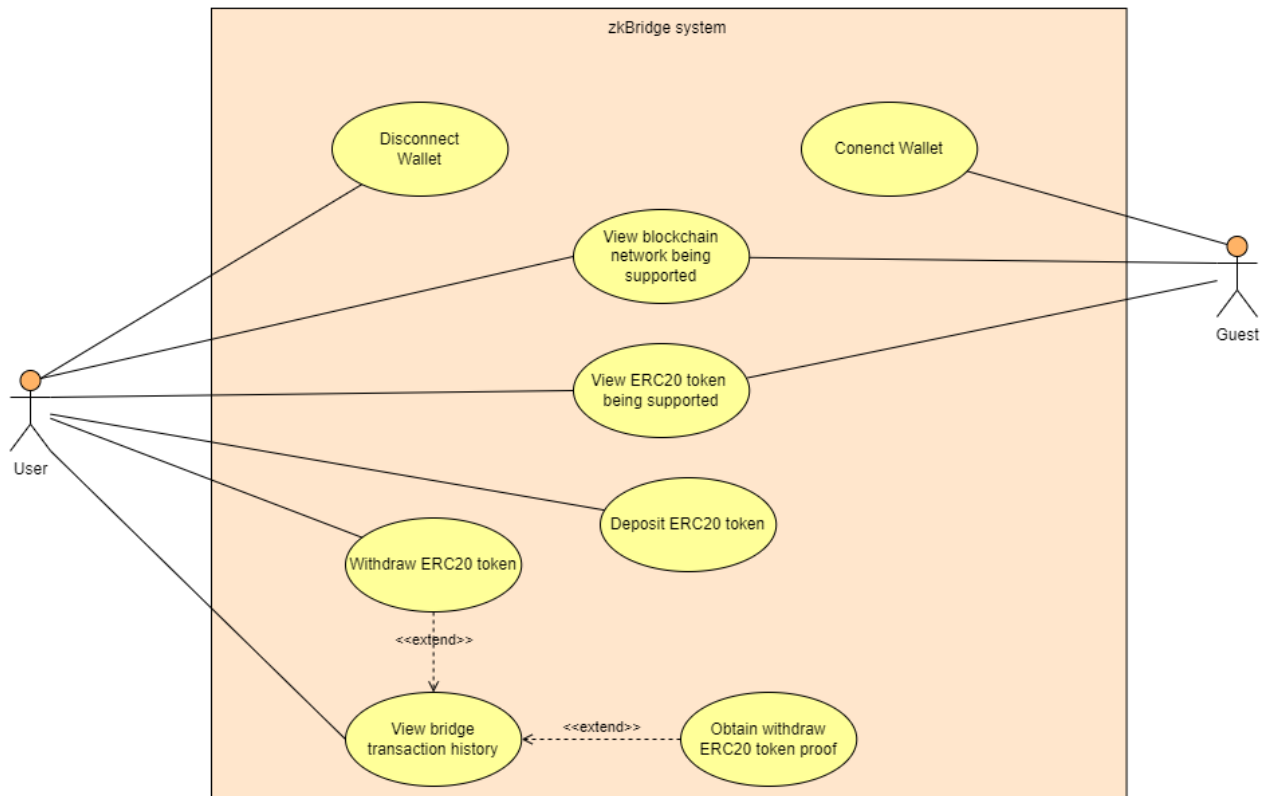


Figure 4.13: General Use Case.

The actors interacting with the system include Guests and Users. Guests become Users after logging in with their wallets. When a Guest accesses the bridge app, they can view the blockchain networks that support cross-chain asset deposits. Additionally, Guests can see the list of supported assets on these blockchains to determine if the bridge is suitable for their needs.

After logging into their account using their wallet, Users can utilize the Bridge app to deposit or withdraw assets across chains. They can also view their transaction history and obtain proof for withdrawing their assets on the destination chain.

b, Detailed Use Case Diagram

Connect Wallet

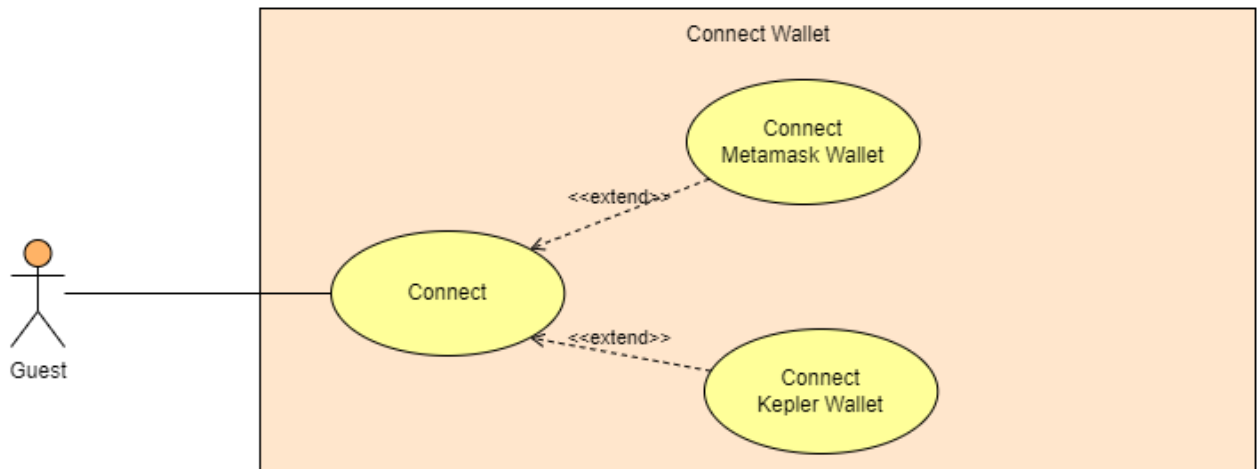


Figure 4.14: Decomposed Use Case: Connect Wallet.

The Guest must connect their wallet to perform transactions. Upon connecting a wallet, they can view the bridge transaction history associated with that wallet.

Disconnect Wallet

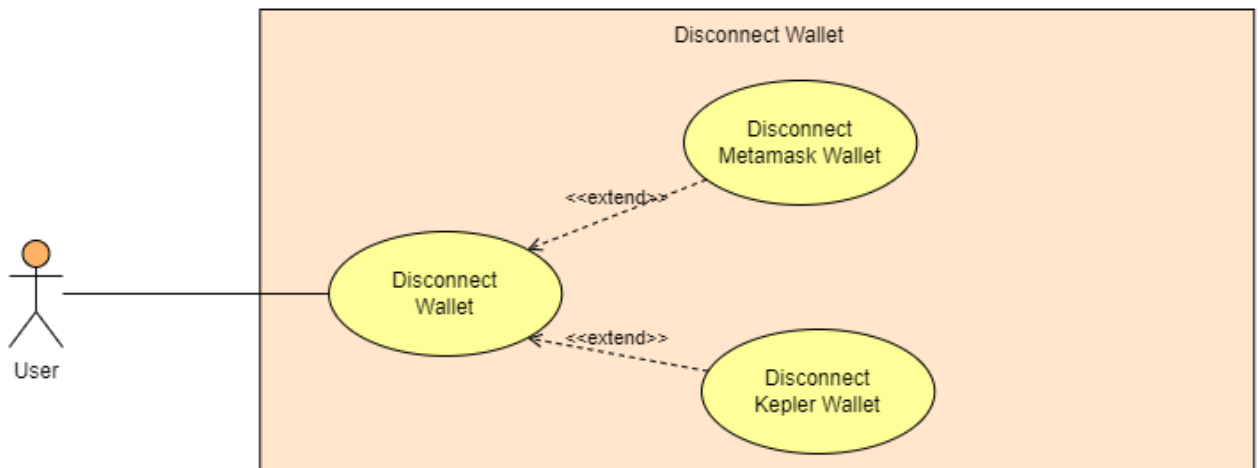


Figure 4.15: Decomposed Use Case: Disconnect Wallet.

In the Use Case: Disconnect Wallet, the User disconnects their wallet and becomes a Guest. However, if they only disconnect one wallet, they can still view their bridge transaction history in the remaining wallet.

Deposit ERC20 Token

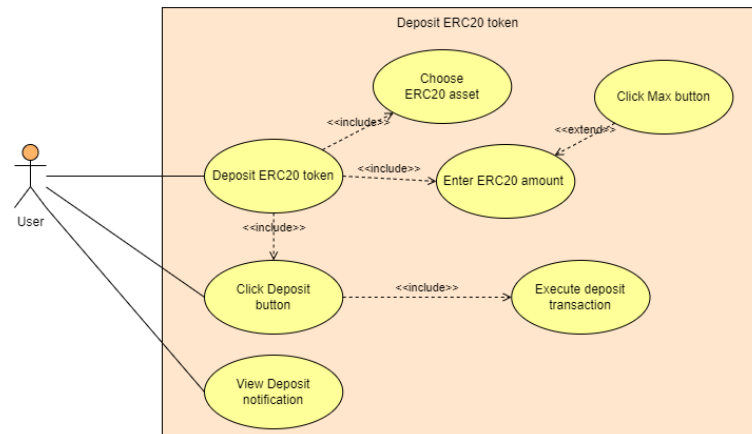


Figure 4.16: Decomposed Use Case: Deposit ERC20 Token.

The Use Case: Deposit ERC20 Token encompasses the lower-level use cases that Users execute when depositing assets across chains. The User selects an ERC20 asset, enters the desired deposit amount, and clicks the Deposit button to initiate the deposit transaction. If the deposit is successful, they can view the deposit notification within the bridge app.

Withdraw ERC20 Token

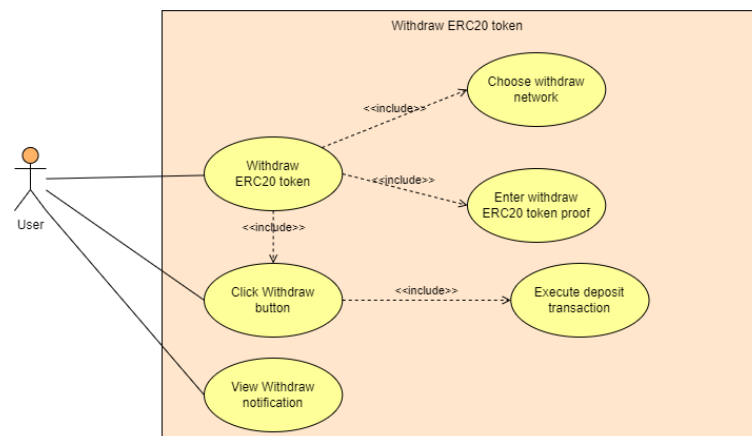


Figure 4.17: Decomposed Use Case: Withdraw ERC20 Token.

To withdraw their deposited assets, Users select the blockchain network where they originally made the deposit (the source chain). They also need to provide proof of the deposit before clicking the Withdraw button to initiate the withdraw transaction. Upon completion, they can view the withdraw notification within the bridge app.

c, Business process

Deposit ERC20 Token

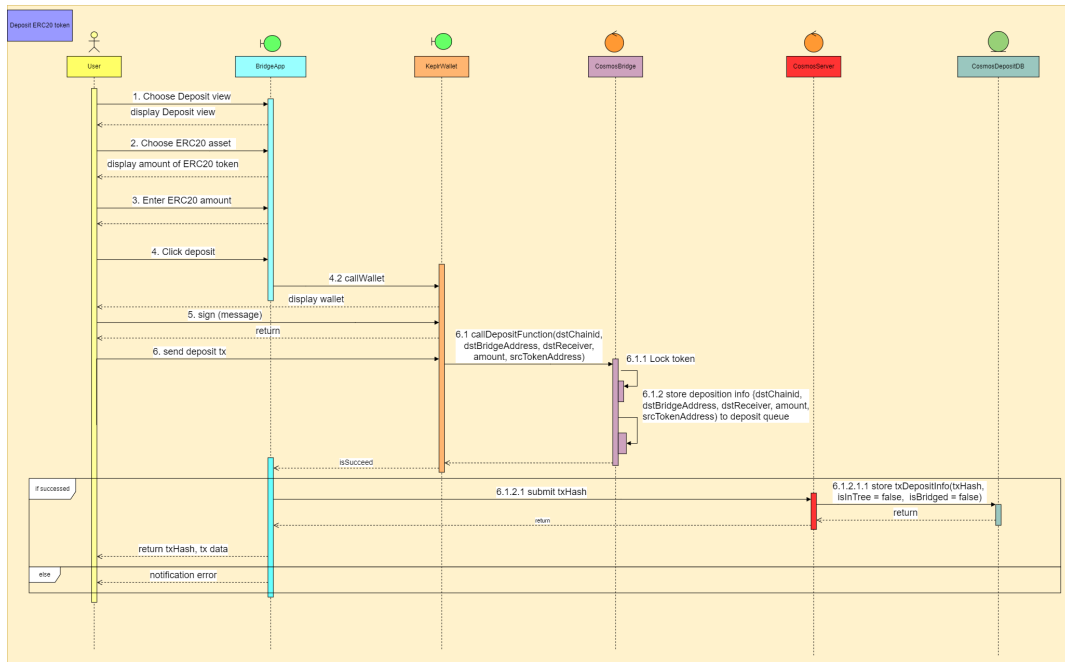


Figure 4.18: Sequence diagram Deposit ERC20 Token.

Withdraw ERC20 Token

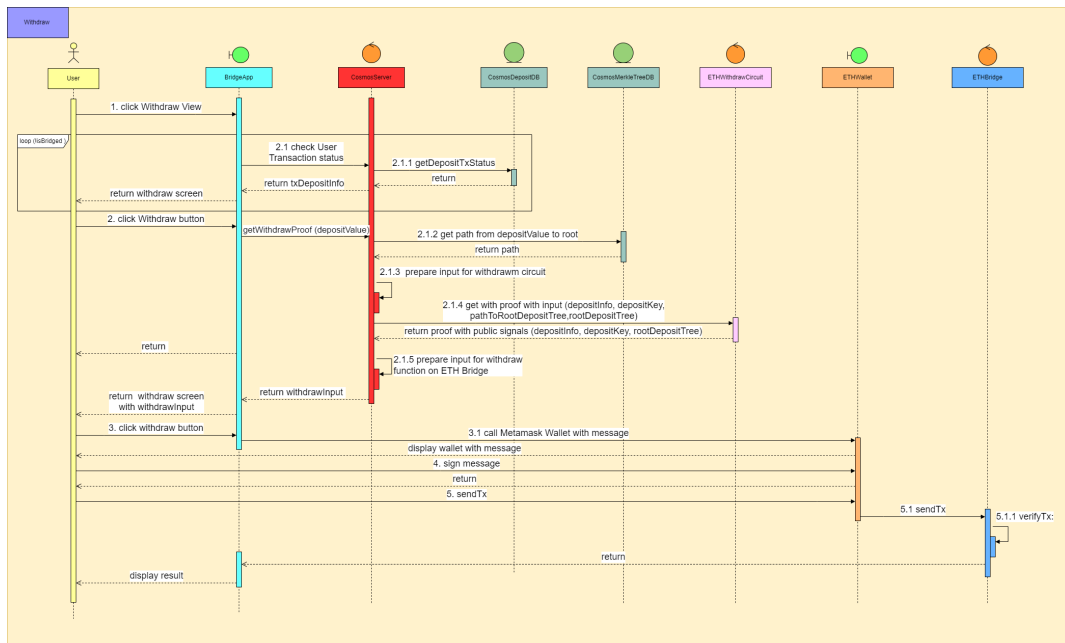


Figure 4.19: Sequence diagram Withdraw ERC20 Token.

4.5.2 Functional description

a, Description of use case Depsit ERC20 Token

Use case code	UC001	Use case name	Deposit ERC20 Token
Actor	Usrer, System		
Precondition	Connect wallet		
Main flow	No.	Executor	Action
	1.	User	Choose Deposit view
	2.	System	Display Deposit view
	3.	User	Choose ERC20 asset
	4.	System	Display amount of ERC20 token
	5.	User	Enter ERC20 amount
	6.	System	Check valid amount
	7.	User	Click Deposit button
	8.	System	Display Keplr Wallet with deposit information
	9.	User	Sign deposit information
	10.	User	Execute deposit transaction
	11.	System	Call Cosmos bridge contract
	12.	System	Store information of deposit transaction
	13.	System	Display notification deposit success with transaction hash and transaction data
Alternative flow	No.	Executor	Action
	6a.	System	Display error: The amount amount is not enough
	8a.	User	Approve their amount asset
Postcondition	None		

Table 4.1: Sepecific Use case Deposit ERC20 token

No.	Attribute	Description	Require	Example
1	Asset type	The type of asset is supported	Yes	Orchai
2	Amount	The amount of asset is chosen want to deposit	Yes	100

Table 4.2: Specific attributes is in UC Deposit ERC 20 Token

b, Description of use case Withdraw ERC20 Token

Use case code	UC002	Use case name	Deposit ERC20 Token
Actor	Usser, System		
Precondition	Connect wallet		
Main flow	No.	Executor	Action
	1.	User	Choose History transaction view
	2.	System	Check User Transaction status
	3.	System	Display History transaction screen
	4.	User	Click Withdraw button
	5.	System	Prepare withdraw input
	6.	System	Display Withdraw screen with withdraw input
	7.	User	Click Withdraw button
	8.	System	Display Metamask Wallet with deposit information
	9.	User	Sign withdraw information
	10.	User	Execute withdraw transaction
	11.	System	Call Ethereum bridge contract
	12.	System	Store information of withdraw transaction succed
	13.	System	Display notification withdraw success
Alternative flow	None		
Postcondition	None		

Table 4.3: Sepecific Use case Withdraw ERC20 token using Hitory view**4.5.3 Non-functional requirement**

After completing the deposit transaction, the user is expected to receive their proof within a maximum of one day. It is important to note that as the number of users using the bridge increases, the time to receive the withdraw proof may be reduced.

CHAPTER 5. EXPERIMENT AND EVALUATION

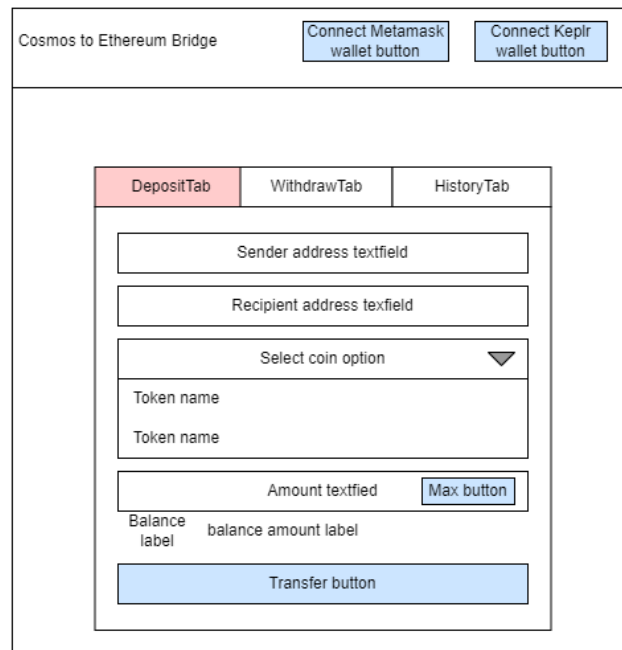
5.1 Detailed design

5.1.1 User interface design

The user interface of the bridge system features a soothing blue color scheme to create a comfortable visual experience for users. Drawing inspiration from other bridge products in the market, the header prominently displays buttons to connect users' wallets, ensuring a familiar and seamless onboarding process.

The main body of the interface is organized into three tabs, catering to the three primary functions: **Deposit Asset**, **Withdraw**, and **History**.

The **Deposit Asset** tab closely resembles other bridge interfaces in the market, comprising four fields: **Sender**, **Receiver**, **Token Type**, and **Amount**. Upon connecting their Kepler and Metamask wallets, the **Sender** and **Receiver** fields are automatically populated, streamlining the deposit process for users. To select the token they wish to deposit, users can use the **Select Token** button. Additionally, a label dynamically displays the maximum balance of the selected token in the user's Cosmos address. For added convenience, a blue **Max** button within the **Amount** field allows users to fill in the field with the maximum available balance for the chosen token. Once all required fields are completed, users can initiate the deposit process by clicking the **Transfer** button.



The mock up UI of the Deposit Tab is shown within a window titled "Cosmos to Ethereum Bridge". At the top right of the window are two buttons: "Connect Metamask wallet button" and "Connect Kepler wallet button". Below these is a tabbed interface with three tabs: "DepositTab" (highlighted in red), "WithdrawTab", and "HistoryTab". The "DepositTab" contains the following elements from top to bottom: a "Sender address textfield", a "Recipient address textfield", a "Select coin option" dropdown menu, a "Token name" label followed by a text input field, an "Amount textfield" with a blue "Max button" to its right, a "Balance label" and a "balance amount label" (both in smaller text), and a large blue "Transfer button" at the bottom.

Figure 5.1: The mock up UI of **Deposit Tab**.

In the **Withdraw** tab, users can claim their assets from the Ethereum bridge. The tab consists of nine fields, with three of them designated for the points **pi_a**, **pi_b**, and **pi_c**, while the remaining six serve as public inputs for verifying the claim proof. One of the six public input fields, the Ethereum bridge address, is set as a constant. For ease of use, users can select the token they wish to withdraw by clicking the **Select Token** button within the token field. Once all required fields are filled, users can proceed with claiming their assets by clicking the blue **Claim** button.

The **History** tab provides users with a comprehensive overview of their transaction history. It displays a list of panels, each containing the name of a deposit transaction, a **Claim** button, and a **See Detail** button. The **See Detail** button allows users to view additional information about their transactions through a text field that appears below the panel. For a smooth claiming process, the blue **Claim** button within each panel pre-fills the required information in the **Withdraw** tab. In the event that a user has not connected their wallet, the interface will display a message indicating that no transactions have been found.

Overall, the user interface design offers a seamless and intuitive bridge experience, empowering users to confidently manage their asset transfers between the Cosmos and Ethereum blockchains with ease.

5.1.2 Database design

The database architecture of my zk bridge system encompasses three key databases: Token Pairs, Deposit Tree, and Deposit Transaction. These databases are implemented using MongoDB, a flexible and scalable NoSQL database that efficiently handles large volumes of data and supports complex queries.

- **Token Pairs database:** This database contains information about token pairs on both the Cosmos and Ethereum blockchains. Each entry in the database includes the following fields:
 - Token symbol: The symbol of the token, e.g., "ETH" for Ethereum or "ATOM" for Cosmos.
 - Ethereum token address: The address of the token contract on the Ethereum blockchain.
 - Cosmos token address: The address of the token contract on the Cosmos blockchain.

The Token Pairs database ensures a one-to-one mapping between tokens on the two blockchains and facilitates quick and easy retrieval of token information for seamless token pair management within the zk bridge system.

- **Deposit Tree database:** This database stores crucial information about the deposit tree state on the Cosmos bridge. Each entry in the database includes the following fields:
 - Current deposit tree root: The root hash of the current deposit tree, representing the state of all deposited transactions.
 - Number of leaf nodes in the tree: The count of leaf nodes (deposited transactions) currently present in the deposit tree.
 - Number of leaf nodes in the queue: The count of leaf nodes (deposited transactions) that are still in the queue, awaiting processing.

The Deposit Tree database leverages MongoDB's document-oriented storage to efficiently manage hierarchical data structures, making it a suitable choice for storing and retrieving deposit tree state.

- **Deposit Transaction database:** This database captures the transaction queue state on the Cosmos bridge. Each entry in the database corresponds to a deposit transaction and includes the following fields:
 - isDeposit (for status): The status of the deposit transaction, which can be one of the following values: "in queue," "in tree," "can claim," or "claimed."
 - Ethereum token address: The address of the token involved in the deposit transaction on the Ethereum blockchain.
 - Ethereum bridge address: The address of the Ethereum bridge contract handling the deposit transaction.
 - Amount: The amount of the token being deposited in the transaction.
 - Ethereum receiver: The address of the recipient of the deposited tokens on the Ethereum blockchain.

MongoDB's dynamic schema allows for the flexible representation of transaction data, accommodating changes in transaction status as they progress through the bridge system.

The key of each deposit transaction in the Deposit Transaction database is its index in the deposit queue on the Cosmos bridge. This indexing ensures efficient access and retrieval of specific deposit transactions when processing and managing the bridge operations.

The use of MongoDB as the underlying database technology ensures the reliability, scal-

ability, and performance of the zk bridge system's data management. By leveraging the strengths of MongoDB, the system can effectively handle the complexities of storing and processing deposit transactions and token pairs, facilitating smooth and secure asset transfers between the Cosmos and Ethereum blockchains [Figure 5.2].

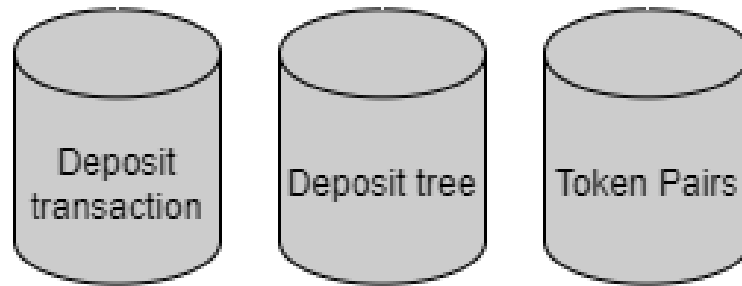


Figure 5.2: The database in zk brige system.

5.2 Application Building

Purpose	Tools/libraries	Reference
IDE	Visual Studio Code	code.visualstudio.com
Database management system	MongoDB	www.mongodb.com
Building interactive UI	ReactJs	reactjs.org
Xây dựng API	ExpressJS	ex[ressjs.com]
Database Management Tools	MongoDB Compass	mongodb/products/compass
Build Cosmos ecosystem	Cosmwasm	docs.cosmwasm.com
Compiling Cosmos contract	Docker	docs.docker.com
Deploying Cosmos contract	Cosmwasmjs	CosmWasm/CosmWasmJS
Checking compile error and test Cosmos contract	Rust-analyzer	rust-analyzer.github.io
Compiling Ethereum contract	Hardhat	hardhat.org
Testing Ethereum contract	Remix	remix.ethereum.org
Interacting with Ethereum contract on Ethereum bridge system	Etherjs	docs.ethers.org
Interacting with Ethereum contract on Application	Web3js	web3js.readthedocs.io
Generating verification key for zk proof	Snarkjs	iden3/snarkjs
Generating zk proof Generating input for circuit	Snarkjs Tendermint, Cosmos-SDK	iden3/rapidsnark tendermint , cosmos-sdk

Table 5.1: List of libraries and tools used

5.2.1 Achievement

Upon completing the zk bridge server for the testnet version, users can now seamlessly bridge their assets in a decentralized manner. The entire process of generating a proof for updating the deposit tree root to the Ethereum bridge system, which involves querying the deposit queue and deposit tree on Cosmos, generating the proof for updating the deposit root on Cosmos bridge, updating the deposit root on Cosmos bridge, waiting for the block header of the update root transaction to be firm and irreversible, generating the proof for updating the block header to the Ethereum bridge, and finally updating the deposit root on the Ethereum bridge, takes approximately two minutes. Additionally, the time required to generate a proof for users to claim their assets is merely one to second. These processes have been efficiently executed on a server with 32GB RAM and a 16-core CPU on Google Cloud. The successful deployment of the testnet version demonstrates that its implementation on the mainnet is entirely feasible. With these capabilities, the zk bridge system empowers users to confidently and securely conduct cross-chain asset transfers, making it a reliable and valuable addition to the blockchain ecosystem.

Below is a table providing additional information about my bridge system. In order to analyze the complexity and size of the codebase, I have implemented a code file specifically designed to count the number of lines of code present in the system. This metric will help in assessing the overall development effort and comprehensiveness of the system's implementation. The table will showcase the breakdown of lines of code for each part of system used in the bridge system, including Cosmos bridge system, Ethereum bridge system, Circuit, server, application.

Product name	Number of lines of code	Number of classes	Number of packages	Size
Cosmos bridge system	3193	18	2	1.4 GB
Ethereum bridge system	4901	150	5	601 MB
Circuit	7093	36	16	7.4 GB
Server	1500	60	16	4 GB
Application	2071	11	2050	1.6 GB
Total	18758	275	50	15 GB

Table 5.2: Statistics my bridge system

5.2.2 Illustration of main functions



Figure 5.3: The UI of **Deposit Tab** after connecting keplr wallet and metamask wallet.

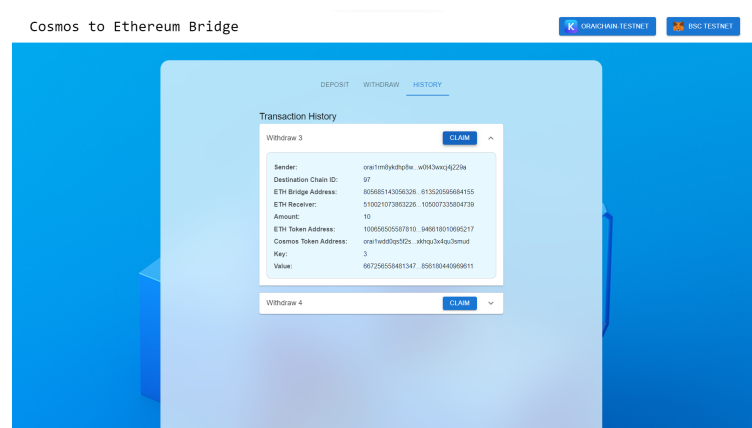


Figure 5.4: The UI of **History Tab** display deposit transaction history of keplr account and meta-mask account.

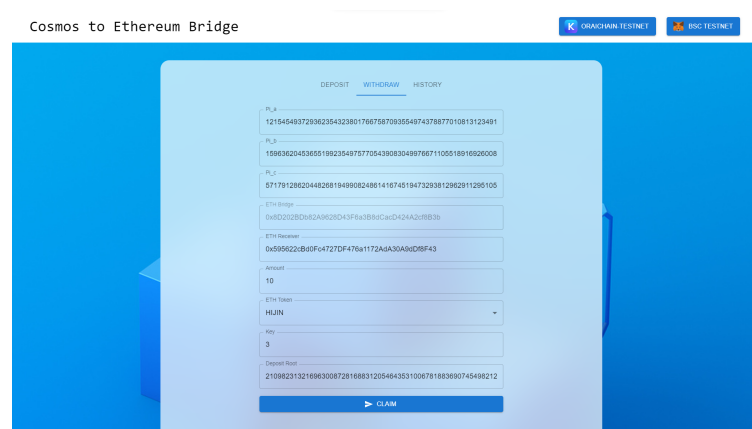


Figure 5.5: The UI of **Withdraw Tab** after click **claim** button in **History Tab**.

5.3 Deployment

In the development and deployment of my bridge system, we utilized a server with 32 GB RAM and a 16-core CPU from Google Cloud, ensuring sufficient computing power to handle the complexities of the bridge operations. my Cosmos bridge system was successfully deployed to the Oraichain-testnet network, which utilizes Cosmos-SDK, providing a robust and feature-rich blockchain environment. Additionally, we deployed my Ethereum bridge system to the Bsc testnet, which is EVM-compatible and offers low gas fees for testing. This testnet allows us to faucet a larger quantity of native tokens compared to other Ethereum testnets like goerli, rinkeby, and sepolia. The deployment and setup of both Cosmos and Ethereum contracts took approximately fifteen minutes. Additionally, generating the zkey file for my circuit, which facilitates zero-knowledge proof generation, required around two hours on the Google Cloud server. However, the deployment of my zk bridge server and application was achieved within a matter of seconds, providing users with quick and seamless access to bridging their assets in a decentralized manner.

CHAPTER 6. MAIN CONTRIBUTIONS

Throughout the product development process, I encountered several challenges that required innovative solutions. Among these, two problems proved to be particularly difficult during the progression of my thesis:

1. The circuit for generating a proof for updating the new Cosmos block header on Ethereum is too large
2. The information from Cosmos network is not compatible for Circom language

6.1 The circuit for generating a proof for updating the new Cosmos block header on Ethereum is too large

6.1.1 Problem

When updating a new Cosmos block header to Ethereum, generating a proof for it [3.2.1](#) becomes a challenging task due to the substantial constraints involved [2.5.5](#). The verification of the Cosmos block header can result in a staggering number of constraints, potentially reaching up to 100,000,000 constraints. This is primarily attributed to the presence of numerous validator signatures, and the verification process for a single signature alone [a](#) requires 2,500,000 constraints. The sheer size of the circuits poses significant limitations on the computational capabilities of the server. Even with a robust server featuring 32GB RAM and 16-core 3.0GHz (AWS c5a.4xlarge instance), it can only generate proofs for circuits with a constraint count lower than or equal to 4,000,000. To handle circuits with such immense constraints, a server configuration substantially higher than the current setup would be required

6.1.2 Solution

To address this problem, I will write several circuit for verification new Cosmos block header instead of using a circuit for generating proof. I have a circuit generating a proof for verification a validator signature and two circuit to verify validator hash, block hash base on datahash based validator set. Because the validator set is quite large [e](#) with virtually 5.000.000 constraints greater than 4.000.000 constraints. Hence, the proof of validator hash it separate to two circuit. Once circuit calculate a left half of validator set, the other calculate a the remain part and verify block hash base on datahash and validator hash. Then, we generate proof for each validator signature and validator left, validator right. I will present it in [Appendix C](#) in detail.

6.1.3 Result

The generating proof for these circuit above and the logic updating new Cosmos block header to Ethereum contract is success. However, because of the number of proof increase after separate circuit, so the gas fee when update a new block header will be a lot with one circuit.

6.2 The information from Cosmos network is not compatible for Circom language

6.2.1 Problem

In the Circom language, it is necessary to configure the logic execution from the beginning. However, in Cosmos, there are many fields in the block header that can vary in byte length, such as the block timestamp. The byte length used to encode the timestamp ranges from 11 to 15 bytes, and it is hashed before being added to an AVL++ tree (Chapter 2, AVL++ Tree) to create the block hash. When attempting to obtain the hash of this field after encoding in Circom, it will be incorrect if the SHA256 library (circomlib) in Circom is used directly. When passing a value to this library for hashing, the length of the parameter needs to be configured beforehand. If Circom is used for hashing a parameter with a length of 15 bytes, it cannot be used for hashing a parameter with a length of 11 bytes. Although Circom supports conditional statements (if-else), they can only be used for configuring the values of parameters from the beginning and cannot be used for input signal parameters. To use conditional statements (if-else) for input signal parameters, they can be divided into multiple cases, but the complexity or content of the circuit will be equal to the constraints in each of the cases. The byte length of the timestamp ranges from 11 to 15 bytes, and if it is divided into cases for processing, it would require 5 cases and the constraints would increase by 5 times to hash this field. Moreover, it is not only the timestamp field that has variable length; there are also many fields in transaction data (Chapter 2, Cosmos Transaction) and the voting power of validators (Chapter 2, Validators) that have variable lengths. Therefore, dividing into many if-else cases for each length is not feasible as it would result in an excessively large circuit constraint, which cannot be handled by a computer with low configuration.

6.2.2 Solution

I have researched the hash mechanism and developed my own library to address this problem. One advantage is that, except for the "msg" field in transaction data, all other fields in the block header, validators, or transaction data that have varying lengths after encoding and padding bits to use the SHA hash function (Chapter 2, SHA) have the same length. Therefore, I will still use the SHA library in Circom for the "Compress function" step. To

hash a parameter with a length that can change each time it is passed, I will first determine its actual length by checking the length of the string of consecutive zero bytes at the end of the passed byte length. The actual length will be equal to the byte length passed minus the length of the string of zero bytes at the end. Once the actual length is determined, I will perform the steps of the SHA hash function in Circom, including the compress function in the Circom library. As for the "msg" field in transaction data, I will create a message in such a way that the byte length of the transaction data, after encoding and padding bits, becomes constant.

Additionally, I have also developed algorithms in Circom, such as concatenating two strings, each with a variable length, and encoding the fields in the block header, transaction data, and validator sets, among others. I will present it in [Appendix C](#) in detail.

6.2.3 Result

My circuit can verify block hash, data hash, and validator hash based on the parameters passed to the circuit without requiring the knowledge of their actual lengths.

CHAPTER 7. CONCLUSION AND FUTURE WORK

7.1 Conclusion

In my project, I have successfully addressed the goals that I declared in Chapter 1 [1.2](#):

1. **Bridging Cosmos block header to Ethereum:** To achieve this, my approach was based on the verification block header mechanism on the Cosmos network of validators [Chapter 3 [3.2](#)]. Based on Tendermint mechanism, I implemented a process to verify the validator signature in the block header and count the total voting power of signed validators. If the total voting power of signed validators was greater than $\frac{2}{3}$ of the total voting power of validators in the validator set, the block header was considered valid. Additionally, I conducted extensive research to understand how the block hash, data hash (the root of transactions in the block header), and validator hash were formed. Using this knowledge, I created a Circom file for verification to generate Zero-Knowledge proofs for verifying the validator signature, block hash, and calculating data hash and validator hash [Appendix C [C.2](#)]. I also designed the Cosmos bridge system to verify these Zero-Knowledge proofs and update the information of the Cosmos block header, including the block hash, data hash, validator set, and validator hash [Chapter 4 [4.3](#)].
2. **Bridging deposit root transaction:** The methodology for achieving this goal was presented in Chapter 3 [3.3](#). Based on Cosmos SDK, my approach involved constructing a deposit tree [Chapter 3 [a](#)] by hashing the information in deposit transactions and using the hashes as values in the tree. After a certain time, I implemented a circuit to verify the new root deposit tree after updating it with new deposit transactions [Appendix C [C.1](#)]. Once verified, I updated the deposit tree root in the Cosmos transaction. To do this, I used the stored deposit transactions and old deposit tree in the Cosmos bridge to verify the Zero-Knowledge proof for the new root deposit tree. Once I confirmed that the updating of the deposit root transaction was irreversible, I sent it to the Ethereum bridge. The deposit root transaction data was hashed to receive a transaction hash, which was then included in the transaction tree with its root as datahash, storing the block header [Chapter 2 [b](#)]. Since the root of the deposit tree is part of the transaction data, I needed to extract it from the transaction data and verify the transaction data by indicating that it would form the data hash that is confirmed and stored in the Ethereum bridge. To accomplish this, I wrote a circuit for verifying the root of the deposit tree [Appendix C [C.2.1](#)] and built a mechanism for updating the deposit tree root on the

Ethereum bridge.

3. **Claiming deposit transaction:** This is the final goal and is discussed in Chapter 3 [3.4](#). To achieve this, I utilized the deposit tree root stored on the Ethereum bridge and wrote a circuit to verify whether a user's deposit root transaction is a part of the deposit tree root [Appendix C ??]. Users can then use this proof to claim their assets that they had deposited on the Cosmos bridge.

Upon completing my thesis, I have gained valuable insights into the challenges I encountered throughout the development process. The scarcity of public resources and reference materials for constructing the bridge demanded significant effort in devising novel approaches and optimizing constraints. Moreover, working with multiple programming languages introduced complexity to the development phase, enabling me to attain proficiency in Solidity, Rust, Golang, Typescript, Python, Circom, HTML, and CSS to establish a robust and fully functional system.

The experience of researching and implementing the bridge emphasized the importance of persistence, adaptability, and resourcefulness when tackling new and intricate problems. The transition from research to production unveiled unique challenges, with limited computer resources posing practical implementation constraints. Ultimately, this journey underscored the symbiotic relationship between theoretical research and real-world application, accentuating the need for innovative problem-solving to surmount obstacles in the realm of cutting-edge technology development.

7.2 Future Work

The current version of the bridge only supports one-way asset transfers, enabling the movement of ERC20 tokens from the Cosmos network to Ethereum. In the future, the bridge will be expanded to facilitate two-way transfers, allowing users to transfer assets between the Cosmos and Ethereum networks in both directions. Additionally, a decentralized mechanism will be developed to update the ValidatorsSet of both networks. This will address scenarios where the number of validators in the new block of the source network significantly deviates from the number of old validators stored on the destination network, rendering the Tendermint mechanism inadequate. Ultimately, the goal is to construct decentralized bridges that facilitate seamless interoperability between various blockchain networks.

The completion of this bridge project marks an important milestone in the advancement

of blockchain technology. The ability to securely and efficiently transfer assets between distinct networks opens up new possibilities for cross-chain collaboration and decentralized finance. The ongoing development and enhancement of bridges will play a vital role in fostering interoperability and realizing the full potential of blockchain ecosystems.

REFERENCE

- [1] T. Xie, J. Zhang, Z. Cheng, *et al.*, “Zkbridge: Trustless cross-chain bridges made practical,” *arXiv preprint arXiv:2210.00264*, 2022.
- [2] circom ecdsa, *Circom-ecdsa*. [Online]. Available: <https://github.com/0xPARC/circom-ecdsa>.
- [3] ed25519 circom, *Ed25519-circom*. [Online]. Available: <https://github.com/Electron-Labs/ed25519-circom>.

APPENDIX

A. THE FUNCTIONS IN COSMOS BRIDGE SYSTEM

A.1 Support token pair

In the Bridge system, each Cosmos token address is associated with a corresponding Ethereum token address, and these two addresses together form a token pair. When integrating a new token into the Bridge system, the required information includes the token addresses of the token pair and its corresponding chainId. By providing this data, the bridge can facilitate the registration of the newly added token pairs. This process ensures seamless support for the token pairs, enabling efficient cross-chain asset transfers and interoperability within the Cosmos ecosystem.

A.2 Receive ew20 token

When users deposit their assets to the Ethereum network, the Cosmos bridge receives the tokens and securely locks them. Additionally, the bridge maintains a deposit queue that stores essential information about each deposit transaction. The information includes:

- **isDeposit:** Indicates whether the deposit transaction has been added to the tree or remains in the deposit queue.
- **sender:** The sender's address of the deposit transaction.
- **destination chainid:** The unique identifier (chainid) of the destination EVM network to ensure proper transaction handling, especially in the presence of multiple EVM blockchains.
- **destination bridge address:** The address of the EVM bridge that allows users to withdraw their deposited assets.
- **receiver:** The receiver's address on the destination chain.
- **amount:** The amount that has been deposited.
- **cosmos token address:** The address of the asset token that has been bridged.
- **key:** The index of the deposit transaction in the queue.
- **value:** The Poseidon hash of the destination bridge address, receiver, amount, and the Ethereum token address paired with the Cosmos token address. This hash represents a leaf in the deposit tree at the key-th position [Figure A.1].

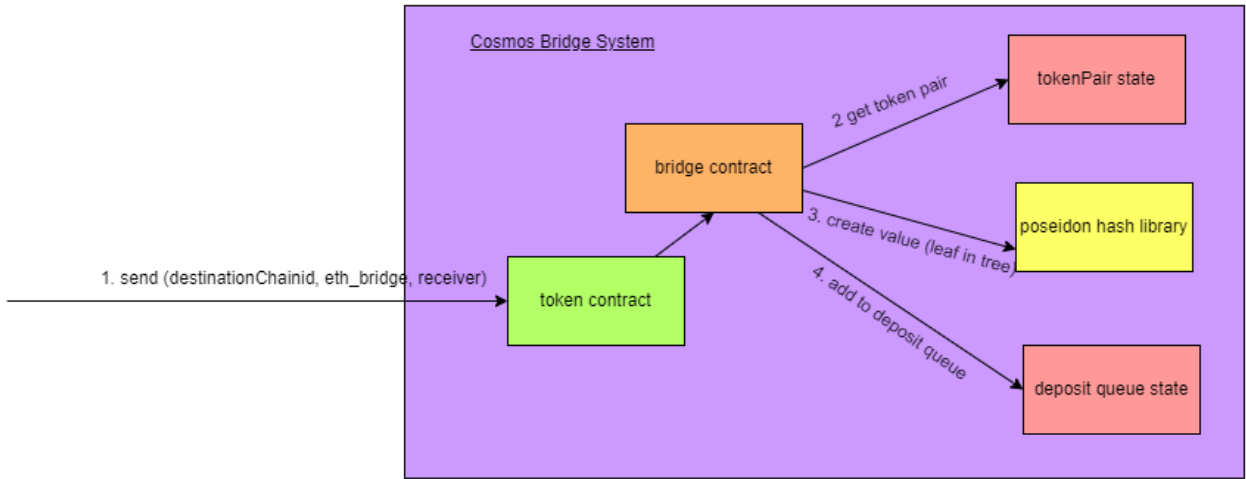


Figure A.1: Receiver function flow in Cosmos bridge.

A.3 Update deposit tree

A.3.1 Update deposit tree mechanism

The deposit tree in the Cosmos contract serves as a repository for essential deposit-related information, including the current deposit root [a](#). This root is computed by aggregating the values of deposit transactions in the deposit queue, utilizing their respective keys. Furthermore, the deposit tree stores the current counts of deposit transactions both in the tree and the queue.

At regular intervals, the server queries the transactions present in the queue. Subsequently, these transactions are incorporated into the deposit tree on the server, leading to the generation of a proof. This proof is then used to update the new deposit tree on the Cosmos Bridge [b](#). Through a thorough verification process, the bridge confirms the validity of the proof and marks the corresponding deposit transactions as added to the tree. Finally, the Cosmos Bridge updates the new deposit root, as well as the counts of deposit transactions in both the tree and the queue.

This seamless process of updating and synchronizing the deposit tree ensures the accuracy and integrity of deposit-related data within the Cosmos ecosystem, facilitating smooth and secure asset transfers between the Cosmos and Ethereum networks [Figure [A.2](#)].

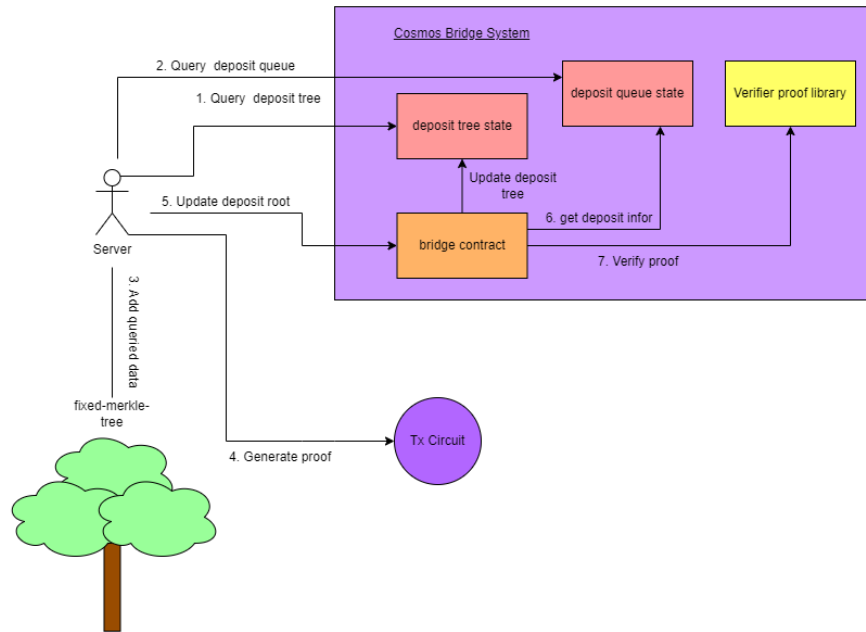


Figure A.2: Update deposit root function flow in Cosmos bridge.

A.3.2 Withdraw ew20 token

When users deposit assets from Ethereum to Cosmos, they utilize a designated function to receive their tokens. However, in the existing version of the bridge, the transfer is unidirectional, allowing assets to move only from Cosmos to Ethereum but lacking the reverse direction. To address this limitation, the asset will be withdrawn from the bridge contract if it possesses the necessary admin signature. This precaution ensures that assets do not become stuck in the Cosmos bridge contract, providing a secure and efficient asset transfer mechanism for users.

B. UPDATER CONTRACTS IN ETHEREUM BRIDGE SYSTEM

B.1 Gate contract

Upon receiving new Cosmos block header information, the gate contract initiates a verification process by utilizing the verifier contract to confirm the validity of the proof. The proof encompasses thorough checks of all validators' signatures, as well as the block hash, data hash, and validator hash. Once the proof is successfully verified, the bridge system proceeds to validate the tendermint mechanism by inspecting the block height and the validator set. After confirmation of the tendermint mechanism's compliance, the block hash and data hash, along with the validator hash, are stored in the block header contract. Concurrently, the validator set is stored in the validator contract [Figure B.1]. Subsequently, the admin sends the deposit root proof to the gate contract. The gate contract diligently verifies the proof and confirms whether the data hash in the deposit root aligns with the data hash stored in the block header contract. Upon successful validation, the gate contract proceeds to store the deposit root in the bridge contract, finalizing the process and ensuring the integrity of the data in the Ethereum bridge system [Figure B.2].

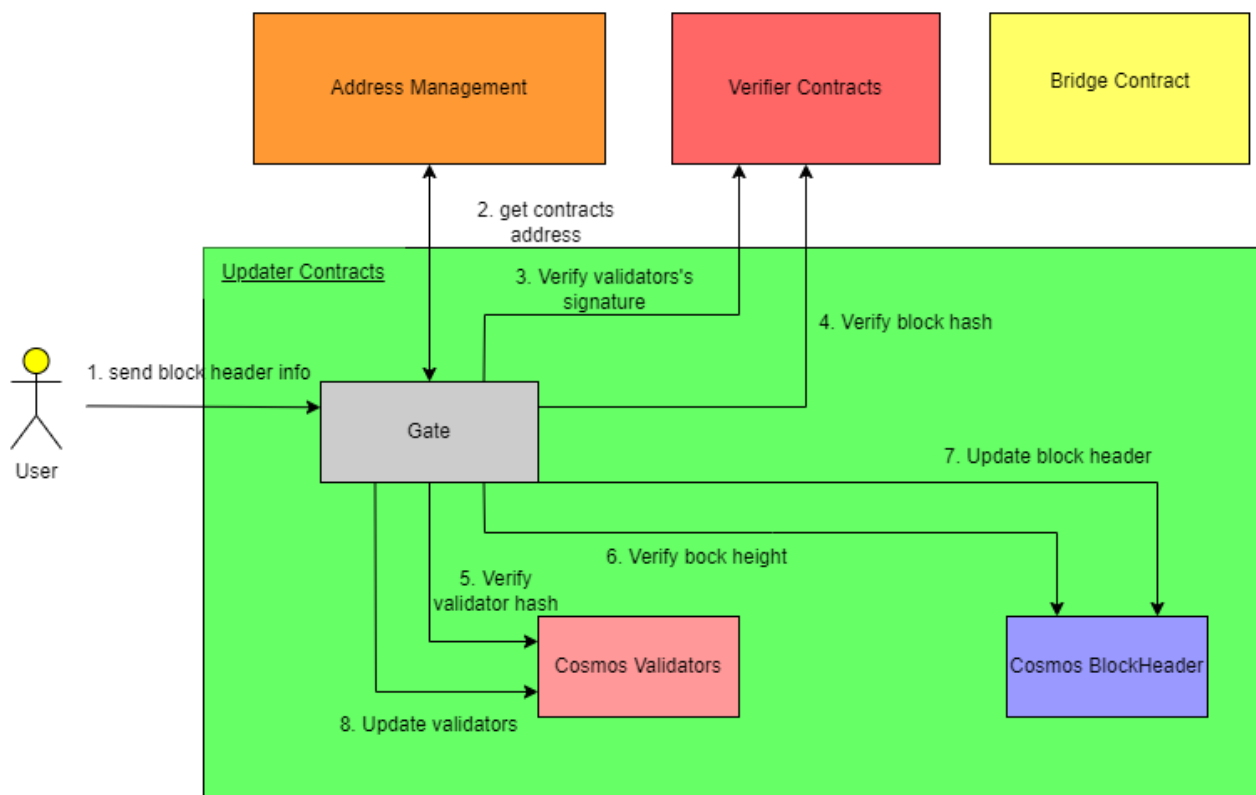


Figure B.1: Update block header flow.

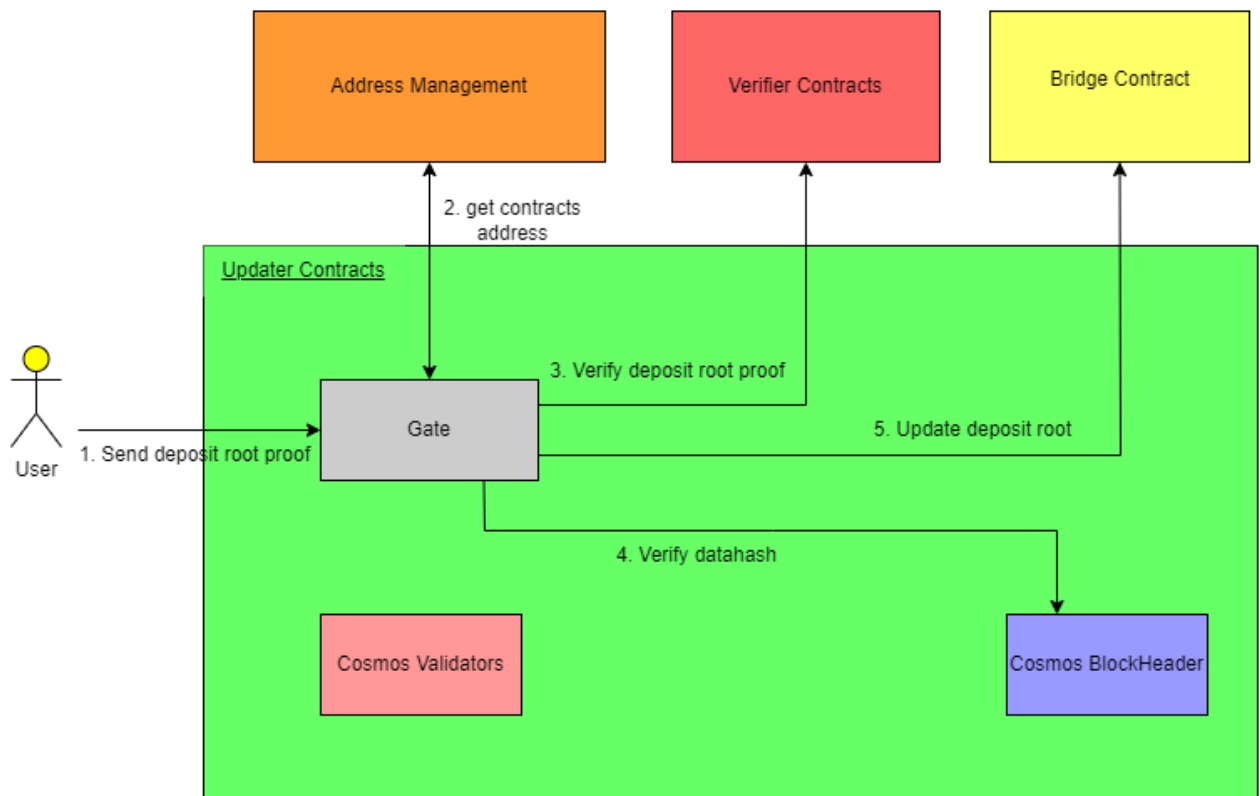


Figure B.2: Update deposit root flow.

B.2 Cosmos block header contract

The Cosmos block header contract plays a crucial role in the Ethereum bridge system by storing essential information related to Cosmos block headers. Only the gate contract is granted permission to update this information. For transparency and accessibility, the Cosmos block header contract makes certain information publicly available to users. This includes the current block height, current block hash, and current data hash. Additionally, the contract provides user-friendly functions, allowing users to retrieve block hash by specifying the block height and obtain data hash by height. By exposing these functionalities, the Cosmos block header contract ensures that users can easily access and verify Cosmos block header information, contributing to the overall transparency and efficiency of the Ethereum bridge system [Figure B.3].

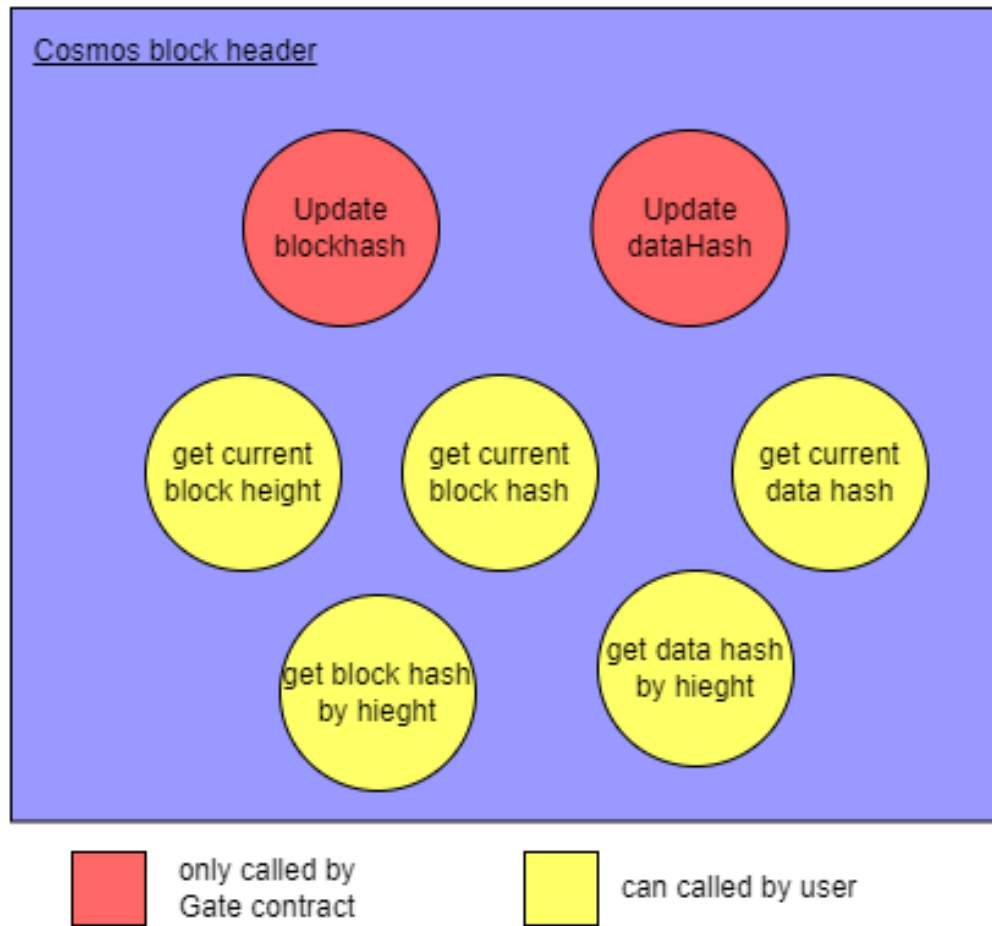


Figure B.3: Cosmos block header functions.

B.3 Cosmos validator contract

The Cosmos validator contract serves a crucial role in the Ethereum bridge system by verifying and storing the new validator set along with its corresponding validator hash. This verification process ensures the integrity and security of the validator set before it is incorporated into the system. Besides, only the gate contract is granted permission to update new validator set. In addition, the contract provides user-friendly functionalities for retrieving essential information. Users can access details such as the current block height and the validator set at a specific height. Moreover, users can query specific validators within the set by their index at a particular height. Additionally, the contract allows users to retrieve the block height associated with a specific validator set. These functionalities enable users to seamlessly access and utilize validator-related information, enhancing transparency and facilitating efficient interactions within the Ethereum bridge system [Figure B.4].

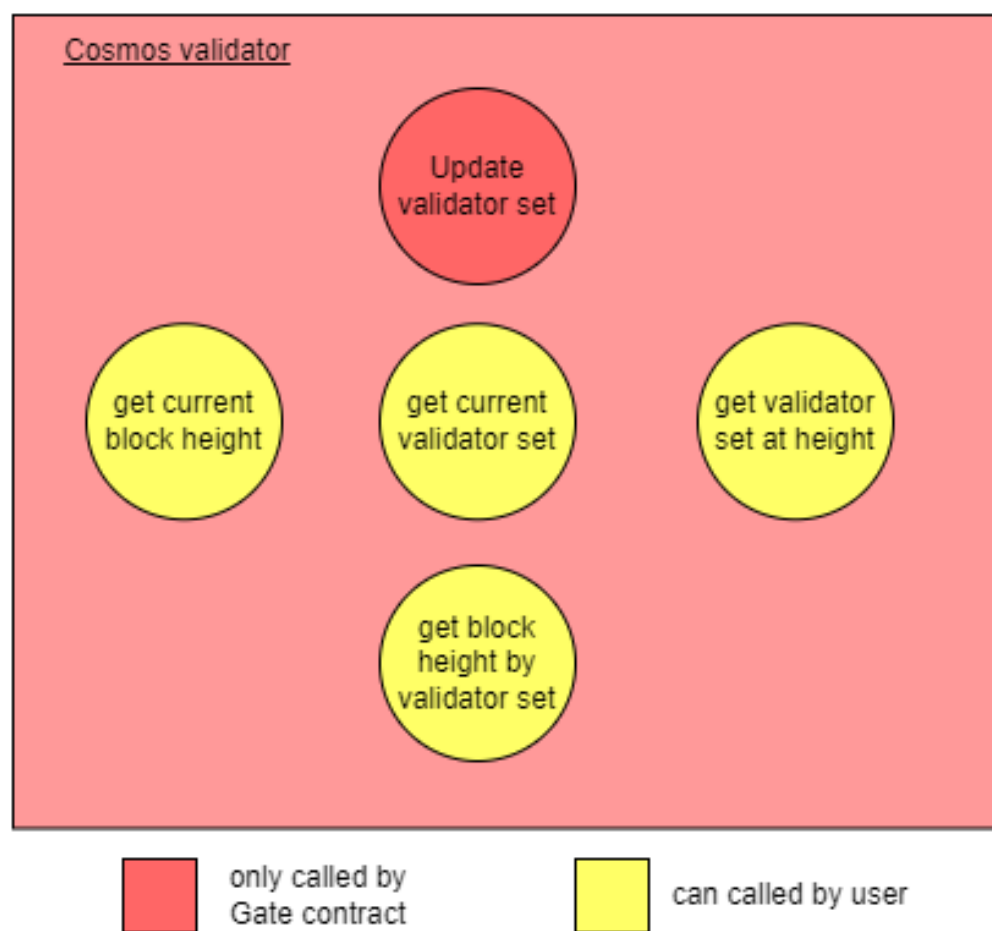


Figure B.4: Cosmos validator functions.

C. ZKBridge Circuit

In my circuit system, we have developed multiple circuits to generate proofs for different essential purposes within the Ethereum bridge system. These circuits serve four primary objectives, each critical for ensuring the system's robustness and security. Firstly, we have circuits dedicated to verifying the new deposit tree root on the Cosmos bridge, confirming the integrity of the deposited assets. Secondly, circuits are tailored to verify the new Cosmos block header on the Ethereum bridge, ensuring the accurate and secure transfer of information between the two blockchains. Additionally, we have circuits specifically designed to verify the new deposit tree root on the Ethereum bridge, facilitating seamless asset transfers between the two networks. Lastly, our circuits include mechanisms to verify claim transactions, ensuring that users can withdraw their assets from the Cosmos bridge after depositing them, and maintaining the trustworthiness of the entire Ethereum bridge system. Through this diverse set of circuits, we ensure the comprehensive verification and secure functioning of the Ethereum bridge for various crucial operations.

The picture below [Figure C.1] demonstrates Circuit package diagram. In my circuit, I has three packages. First, Block header package have a role verify Cosmos block header when transferring it to Ethereum bridge. Then, Transaction packe will process the task relate to transaction such as deposition transaction on Cosmos and Ethereum, and claim transaction on Ethereum. Lastly, Libraries packages supply these function support calculating and verify these information from the remaining package.

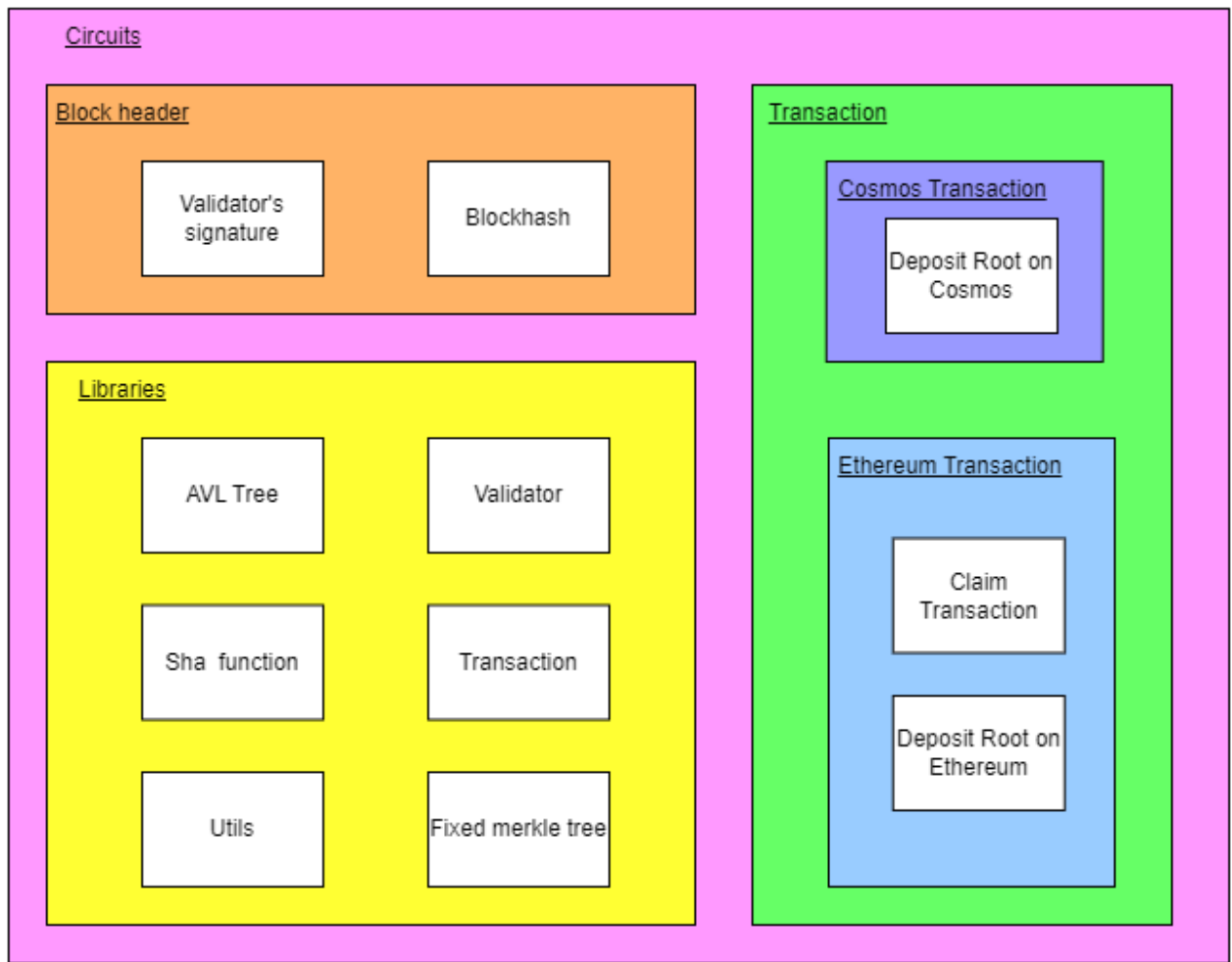


Figure C.1: The package diagram of circuit system.

C.1 Verify new deposit tree root on Cosmos Bridge circuit

In accordance with my proposed methodology [Figure 3.4], I need to develop a circuit to verify the deposit transactions within the deposit tree, structured as a Fixed Merkle Tree. The signal inputs of the circuit encompass the following elements:

- **Key array:** This array comprises the index of the deposit transactions in the deposit queue.
- **Value array:** The new array of values **a** corresponds to the key array in the deposit queue, representing the new leaf of the deposit tree.
- **Old root:** This denotes the root of the deposit tree prior to adding the new leaf array.
- **Siblings array:** This array holds the siblings that correspond with the value array, representing the path from the leaf to the root.
- **New root:** This signifies the root of the deposit tree after incorporating the new leaf

array.

Subsequently, the signal outputs of the circuit include:

- **Key array:** The key array extracted from the signal inputs of the circuit.
- **Value array:** The value array extracted from the signal inputs of the circuit.
- **Old root:** The old root extracted from the signal inputs of the circuit.
- **New root:** The new root extracted from the signal inputs of the circuit.

In my circuit, I aim to prove the correctness of the new root based on the old root when adding a new leaf to the Merkle tree. The initial value of the key before adding a new leaf is the hash poseidon of a zero element, which I have set as a constant variable in the circuit with the value 1901421449564148875923750512694834694297291237961565274103999244586593. Both the old value and the new value at the index key share the same siblings from the leaf to the root. To establish the correctness of the new root after adding a value, I prove that the old value with its siblings can be calculated to the old root, and similarly, the new value with the same siblings can be calculated to the new root. If the old root remains unchanged, it indicates that all the values in the old tree remain intact, thereby confirming that the new root is calculated from the old tree as long as the old root remains unchanged. To prevent any tampering, I store the old root on the blockchain, ensuring its integrity. Whenever a new value is added, and a new root is obtained, I can prove that the new root originates from the old tree. Moreover, when multiple values are added as a batch, I can prove that each new root is correctly calculated after each value addition. For a more detailed understanding, the code examples in the pictures below (Figure C.2 and Figure C.3) illustrate the process of calculating the new root in the circuit.

```
template NewRootTransaction(nSiblings) {
    signal input key;
    signal input oldValue;
    signal input oldRoot;
    signal input newValue;
    signal input siblings[nSiblings];
    signal output newRoot;

    var i;

    component vOld = CalculateRootFromSiblings(nSiblings);
    vOld.key <== key;
    vOld.in <== oldValue;
    for(i = 0; i < nSiblings; i++) {
        vOld.siblings[i] <== siblings[i];
    }
    vOld.root == oldRoot;

    component vNew = CalculateRootFromSiblings(nSiblings);
    vNew.key <== key;
    vNew.in <== newValue;
    for(i = 0; i < nSiblings; i++) {
        vNew.siblings[i] <== siblings[i];
    }
    newRoot <== vNew.root;
}
```

Figure C.2: The picture shows the calculating a new root when adding a new value in circuit.

```

template NewRootBatchTransaction(nTransactions, nSiblings) {
    signal input key[nTransactions];
    signal input oldValue[nTransactions];
    signal input newValue[nTransactions];
    signal input oldRoot;
    signal input siblings[nTransactions][nSiblings];
    signal output newRoot;

    var i;
    var j;

    component verifier[nTransactions];

    for(i = 0; i < nTransactions; i++) {
        verifier[i] = NewRootTransaction(nSiblings);
        verifier[i].key <== key[i];
        verifier[i].oldValue <== oldValue[i];
        verifier[i].oldRoot <== i == 0 ? oldRoot : verifier[i-1].newRoot;
        verifier[i].newValue <== newValue[i];
        for(j = 0; j < nSiblings; j++) {
            verifier[i].siblings[j] <== siblings[i][j];
        }
    }
    newRoot <== verifier[nTransactions - 1].newRoot;
}

```

Figure C.3: The picture shows the calculating a new root when adding a batch of new value in circuit.

C.2 Verify new Cosmos block header circuit

In the process of verifying the Cosmos block header [3.1](#), it is essential to verify each validator's signature in the header as well as the block hash that has been signed by the validators. To accomplish this verification within the circuit, certain information is required. This includes the BlockID [a](#), the height of the block, the public key of each validator, the timestamp signed by the validators, and the signature of each validator. All of this information is encoded into a message using the same encoding mechanism as Tendermint within the circuit.

To achieve this encoding, I referred to the source code of Tendermint and re-implemented it in the Circom language. Though not overly complex, this task demanded considerable time for thorough testing and debugging. Subsequently, to verify the validators' signatures,

which are signed using the Ed25519 signature scheme, I utilized the Circom library provided by Electron-Labs [a](#). However, I encountered an issue with the Circom library, as it only supports verifying messages with fixed lengths. The message encoded from the signed information by validators does not maintain a fixed length across different block headers. Consequently, the encoded message ends up with some zero values between two fields of signed information. To overcome this, I devised an algorithm to concatenate two strings while handling zero values by pushing them to the end of the concatenated string. However, since the actual length of each string is not fixed, the length of the concatenated string varies depending on the input string, making it challenging to determine the exact number of zero values that need to be pushed to the end. As a result, I can only return the concatenated string with zero elements pushed to the end, which results in the encoded message not being an accurate representation, with zero values at the end. As a workaround, I read the Ed25519 mechanism [2.2.2](#) and made adjustments to the Electron-Labs library to enable the verification of this encoded message.

During the verification of all validator signatures in the block header, an important aspect is proving the accuracy of the validator set. This involves calculating the validator hash, which constitutes a part of creating the block hash that is signed by validators [3.2.1](#). To compute the validator hash, the voting powers of all validators are needed. Each validator's public key and voting power are encoded and hashed as a leaf in an AVL+ tree [2.4.1](#). However, the encoded value cannot be hashed directly and requires a multi-step process similar to the message of validator signatures. In my circuit, I also construct an AVL+ tree to calculate the root tree, simulating the tendermint process, and I utilize SHA256 and SHA512 functions from the Iden3 organization. With this setup, I can calculate the validator hash from the validators' public keys and their voting powers.

To prove that the validator hash is a part of the block hash, I incorporate additional information from the block header. Thankfully, the datahash and validatorhash positions in the block header are the 7th and 8th positions, respectively, out of the 14 attributes [a](#). These 14 attributes serve as leaves of the block tree, with the tree root being the block hash. As a result, validatorhash and datahash are adjacent siblings. Instead of using all 14 attributes, I only require datahash, validatorhash, and their parent's siblings to achieve the desired proof.

When it comes to verifying the block hash, I can compile the circuit to generate a zkey file. However, the circuit size becomes a challenge as the constraints for each time verifying the validator signature reach nearly 2,650,000 constraints, which includes encoding the message and verifying the signature on the message [2.5.5](#). Given that Oraichain network

has more than 50 validators, the circuit size exceeds 100 million constraints, surpassing the computational capacity of my server cloud (32GB RAM and 16 cores). My server is unable to handle circuits with nearly 8 million constraints. However, for Oraichain-test, which only has one validator, I can design a circuit that verifies both the signature and block hash in one circuit with approximately 3,900,000 constraints.

In the process of verifying the Cosmos block header on Ethereum, I have devised two solutions: one for the testnet and another for the mainnet, which features a larger number of validators. For the testnet solution, as presented earlier, I generate a proof for each validator signature using the validator's public key as a public input. Additionally, I have developed a separate circuit to verify the block hash, which takes public key, voting power, data hash, and the siblings of the parent of the datahash and validator hash as individual inputs. A Boolean array named "isSigned" in the circuit indicates whether a validator in the validator set has signed the block header or not. By utilizing this array, I can check the validity of the total voting power for signed validators.

However, the constraints for this circuit become substantial, particularly due to the usage of the sha256 function, which accounts for nearly 50,000 constraints. To compute the validator hash from a validator set containing 50 validators, it requires at least 100 hash calculations. Considering the time taken for verifying the block hash, the circuit size grows to nearly 8,000,000 constraints, exceeding the processing capabilities of my server.

To overcome this challenge, I decided to split the circuit into two separate files. One file calculates the root for the first 32 validators in the validator leaf on the left side, producing a value known as the **left child root**. The other circuit includes the remaining validators and the information needed to verify the block hash. These remaining validators are used to compute the right child root (similarly to the left child root but for the 32nd index). The circuit takes the left child root as a signal input and verifies the validator hash. Additionally, because the number of validators on the right side is smaller, this circuit also serves to verify the block header. The constraints for the left and right circuits are approximately 3,900,000 and 4,000,000, respectively.

Once I have all the proofs for the block header, especially for networks with a large number of validators like Oraichain, I cannot update all of them to the contract simultaneously due to the parameter limit in Solidity functions. Consequently, my Ethereum bridge contract will feature two functions for verifying the left circuit with validator signatures on the left and the right circuit with validator signatures on the right. This approach allows for the efficient handling of proofs for block headers with a substantial number of validators.

C.2.1 Verify new deposit tree on Ethereum bridge

Based on the methodology presented in Chapter 3 regarding the mechanism for transferring deposit transactions with the deposit root, I needed to develop a circom circuit to verify that the update root transaction data is a part of creating the data hash, which represents the root of transactions in the block header stored on the Ethereum bridge during bridge block header updates. The deposit root serves as an essential component of the **msg** fields within the **Body** of the transaction data [b](#). Therefore, in the circom circuit, I extracted the deposit root from the update root transaction data. To verify the transaction data, I constructed a circom circuit based on the Cosmos SDK [f](#), which encodes all the fields in the transaction data into a message. The message is then hashed to obtain the transaction hash, representing a leaf in the transaction tree. The mechanism for verifying datahash from transaction data is similar to verifying validator hash from the validator set, and handling these leaves in both cases was relatively straightforward.

However, the transaction data includes numerous fields, leading to a substantial number of constraints when attempting to concatenate all of them. To overcome this issue, I realized that the only field I needed to focus on was the deposit root within the **body** part. To avoid encoding unnecessary data, I incorporated the body, auth info, and signature as signal inputs. I then extracted the deposit root from the encoded body. To achieve this, I designed the structure of the *msg* that I would execute on the Cosmos bridge for the deposit root. Aside from the *msg* field, the other fields in the body part were arranged in a fixed-length manner when encoded. As a result, before encoding the *msg* and appending it to the end of the encoded body, the other fields were encoded, ensuring that the length from the beginning of the encoded body to the beginning of the encoded *msg* was constant. To facilitate the extraction of deposit root information in the encoded body, I set the deposit root as the first parameter of the update deposit root function in Cosmos. The constraint for this circuit is nearly 1,000,000 constraints, making it computationally manageable.

C.2.2 Verify claim transaction

?? The verification of a value below a tree is one of the easiest parts in my Circuit system. Unlike other circuits, this specific circuit requires only a simple indication that there exists a path from the value to the root of the tree. If such a path exists, it proves that the value has been added to the tree. The implementation of this circuit involves a relatively small number of constraints, totaling 8041 constraints. This straightforward verification process ensures the integrity and accuracy of the tree structure, contributing to the overall reliability and efficiency of the zk bridge system.

D. SERVER API

D.1 Deposit transaction api

The Deposit Transaction API plays a crucial role in updating the deposit transaction root on both the Cosmos and Ethereum bridges. It also facilitates users in receiving their claim proofs for claiming their assets. The API encompasses six main endpoints:

- **POST /api/query:** This endpoint allows the query of deposit transactions and deposit tree data on the Cosmos bridge. When a new deposit transaction is identified, it is stored in the database for future reference.
- **DELETE /api/delete:** This endpoint enables the deletion of deposit transactions and deposit tree data from the database.
- **POST /api/update:** The core functionality of this endpoint is to update the deposit root on the Cosmos bridge and then subsequently on the Ethereum bridge. To achieve this, the API first queries deposit transactions and deposit tree data, adding them to the database if necessary. It then proceeds to generate a proof for updating the deposit root on the Cosmos bridge [C.1](#). Following this, the server retrieves the transaction hash associated with the deposit root update and queries the relevant block header and transaction data. The server waits for confirmation, ensuring the block header of the deposit transaction is irreversible (approximately 40 seconds or six blocks). Once confirmed, the server prepares the necessary input for generating the block header proof on the Ethereum bridge. This input processing involves utilizing the tendermint library in Go for Cosmos block header information and the Electron-Labs library to verify Ed25519 signatures in Circom. Additionally, Python and JavaScript are used to process and format data before generating the final input. With the input ready, the server generates the proof for updating the new block header on the Ethereum bridge. Subsequently, the server prepares input for generating the deposit root proof on the Ethereum bridge. Interacting with the Cosmos SDK using Go and JavaScript, the server retrieves the update transaction data and generates the deposit proof, updating it on the Ethereum bridge.
- **GET /api/proof/key:** This endpoint allows users to obtain the proof based on the index of the deposit transaction, facilitating their claim process.
- **GET /api/infor:** Through this endpoint, users can access the list of deposit transaction history based on either their Cosmos sender address or Ethereum receiver address.

D.2 Token pair api

The Token Pair API plays a critical role in the management of token pairs within the zk bridge system. It serves as an interface that allows the front end to query token information, enabling seamless interactions for users. Additionally, when a new token seeks to participate in the bridge system, its data is stored in the Token Pair database through this API. The Token Pair API comprises three endpoints:

- **POST /tokenPairs:** This endpoint receives the addresses of tokens in both the Cosmos and Ethereum blockchains, along with their respective symbols, as parameters. It then stores this information in the Token Pair database, ensuring a one-to-one mapping between tokens on the two blockchains.
- **GET /tokenPairs:** By accessing this endpoint, users can retrieve a list of token pairs or query a specific token pair by providing its symbol, Ethereum token address, or Cosmos token address as parameters. If no specific query is provided, the API returns the complete list of token pairs.
- **DELETE /tokenPairs:** This endpoint allows the deletion of a token pair from the database based on its symbol, Ethereum token address, or Cosmos token address. However, to prevent accidental data loss, the API does not permit the deletion of all token pairs simultaneously.