HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

GRADUATION THESIS

A social login solution for Web3 using Shamir's secret sharing and verified DKG

NGUYEN TUAN MINH

minh.nt184294@sis.hust.edu.vn

Major: ICT Global

Specialization: Information Technology

Supervisor: Ph.D Dao Thanh Chung ____________________

Signature

Department: Computer Engineering

School: Information and Communications Technology

HANOI, 06/2022

Requirements for the thesis

Student information

Student name: Nguyen Tuan Minh
Tel: 0915871399 Email: minh.nt184294@sis.hust.edu.vn
Class: ICT02.K63 Program: Global ICT

This thesis is performed at: 21st floor, Charmvit Tower

Goal of the thesis

This thesis focus on addressing the challlenges associated with decentralized identity and authentication in blockchain applications, providing developers with a convenient and standardized way to implement secure and user-friendly authentication mechanism.

Main tasks

In this thesis, I will disscuss blockchain, smart contracts, and social login for Web3 Application. Next, I will describe in detail the architecture and design of the Social login system using Shamir's secret sharing and verified DKG. Lastly, i will conduct some experiments to evaluate and querying the efficacy of the solution.

Declaration of student

Nguyen Tuan Minh - hereby attests that the work and presentation in this thesis were carried out by myself under the direction of Ph.D Thanh-Chung Dao. All results presented in this thesis are authentic and have not been plagiarized. All references in this thesis, including images, tables, figures, and quotations, are cited in the bibliography in a plain and comprehensive manner. I will assume full responsibility for any copy that violates school regulations, even if it is only one.

Advisor's confirmation of the completion and defense permission

Hanoi, Ngày 25 tháng 7 năm 2023
Advisor's signature

Dr.Thanh-Chung Dao

ACKNOWLEDGMENTS

I would like to express my profound appreciation to my family, friends and significant other for their unwavering support and patince throughout the process of writing my thesis. Their love, encouragement, and confidence in my abilities have been an inexhaustible source of fortitude and inspiration for me.

Thank you for always being there for me, providing me with a nurturing enviroment, and teaching me the importance of perserverance and diligence. Your unconditional affection and encouragement have inspried me to pursue my academic objectives.

Thank you for solid friendship and for being an unending source of motivation. During the difficult times of thesis writing, your presence, laughter, and words of encouragement have brought me pleasure and helped me maintain a healthy work-life balance.

Lastly, i would like to express my sincerest gratitude to my supervisor, Dr. Dao Thanh Chung. Your direction, expertise and commitment have been indispensable to my research and academic development. Your guidance has not only increased my expertise in the field, but has also inspired me to achieve new heights of intellectual inquiry. Even when the research appeared daunting, your perseverance, encouragement, and unwavering faith in my ability propelled me forward. I am extremely appreciative of the opportunities you have afforded me and the invalueable lessons I have gained under your direction. Thank you for being an outstanding mentor and for your unwavering support throughout the process of writing my thesis.

ABSTRACT

The blockchain has emerged as a revolutionary technology with the potential to transform numerous industries by providing a decentralized and transparent platform for recording transactions and data securely. The administration of identities and authentication remains a significant challenge within the blockchain ecosystem, despite its many benefits. In order to resolve this issue, it is necessary to create software that bridges the gap between conventional web authentication methods and blockchain-based systems. This bridge software would facilitate a more user-friendly and accessible blockchain ecosystem, ensuring that users can access blockchain-based services and applications with seamless identity verification. Blockchain is renowned for its rigorous security features, and any software implementation must maintain this level of security while integrating with standard web authentication protocols. A failure to adequately resolve security concerns could undermine the trustworthiness of blockchain technology. Innovative approaches, such as Shamir's Secret Sharing[1] (SSS) and Distributed Key Generation[2] (DKG), have considerable potential for addressing these issues. SSS is a cryptographic technique that divides a secret into multiple portions before distributing them to participants. This strategy ensures that no single entity has complete access to the secret, thereby enhancing security and reducing the likelihood of unauthorized access. DKG enables the collaborative generation of cryptographic keys without requiring a singular trusted party. This distributed method adds another layer of security and decentralization to the authentication procedure. I intend to develop a social authentication solution for decentralized applications (DApps) using SSS and DKG techniques. This solution would allow users to authenticate using their social network accounts while assuring their privacy and security through the use of secure and distributed authentication protocols. I will design the system architecture, implement the required software components, and assess the solution's performance and efficacy.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

BIP	Bitcoin Improvement Proposal
CRUD	Create, Read, Update, Delete
Dapp	Decentralize application
DeFi	Decentralize finance
DKG	Distributed Key Generation
encKey	Encrypt Key
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
IaaS	Infrastructure as a Service
OAuth2	Open Authenticate 2.0
PoS	Proof of Stake
PoW	Proof of Work
SSS	Shamir's secret sharing

CHAPTER 1. INTRODUCTION

1.1 Motivation

Blockchain technology, as exemplified by Bitcoin [3] and Ethereum [4] networks, has experienced significant growth and garnered widespread attention due to its unique characteristics and prospective benefits. Blockchain has revolutionized many industries, including finance, supply chain, healthcare, and more, by providing a decentralized, transparent, and immutable platform for record-keeping and value transmission. However, conventional web technologies and systems offer their own set of benefits and advantages. Bridging the gap between blockchain and conventional web technologies can unleash a wealth of opportunities and synergies, resulting in a more robust and adaptable digital ecosystem.

One of the key benefits of blockchain technology lies in its ability to provide trust and transparency. The Bitcoin network, for instance, enables peer-to-peer transactions without the need for intermediaries, fostering trust among participants and reducing transaction costs. Ethereum, on the other hand, extends blockchain capabilities by supporting programmable smart contracts, enabling decentralized applications (DApps) with a wide range of use cases. Meanwhile, traditional web technologies offer a well-established infrastructure, user-friendly interfaces, and extensive compatibility with existing systems. By combining the benefits of both blockchain networks like Bitcoin and Ethereum and traditional web technologies, we can create a powerful hybrid solution that leverages the transparency of blockchain while maintaining the usability and familiarity of the traditional web. By facilitating self-sovereign identities and data control, blockchain technology promotes decentralization and empowers individuals. Users can have ownership and control over their digital assets and personal data, decreasing their dependence on centralized entities. This paradigm shift is facilitated by Bitcoin's decentralized network architecture and Ethereum's decentralized application platform. Traditional web technologies, on the other hand, provide users with convenience and familiarity via centralized authentication systems, social logins, and widespread standards. As demonstrated by Bitcoin and Ethereum, integrating these features into the blockchain ecosystem can improve user experience, encourage adoption, and bridge the gap between conventional web users and blockchain applications.

The growth and benefits of blockchain technology, exemplified by networks like Bitcoin and Ethereum, combined with the advantages of traditional web technologies, highlight the importance of bridging the gap between the two. By leveraging the strengths of both systems, we can create a hybrid solution that harnesses the transparency, security, and decentralization of blockchain while maintaining the usability, compatibility, and familiarity of the traditional web. This convergence unlocks new possibilities, expands the reach of blockchain applications, and paves the way for a more interconnected and inclusive digital future. Consequently, the objective of this thesis, a social login solution for Dapps using SSS and verified by DKG, is to combine the advantages of blockchain technology and conventional web authentication.

1.2 Contributions

Due to the fact that this solution is a large undertaking involving the implementation of numerous modules by numerous individuals, it is evident that I did not design and construct the system alone and that other developers participated in its creation. In addition, I was responsible for devising and implementing the mechanisms for the executors to share secrets and generate private keys for end users. I implement the majority of the project's features, with the exception of Shamir's algorithm for sharing secrets and the Distributed Key Generation protocol. In addition, I designed and implemented the majority of the data structures contained in smart contracts and the decentralized storage called Eueno. In addition, this system has a unique architecture for securing and enriching the user experience, as well as enabling developers to integrate existing Dapps seamlessly.

1.3 Thesis structure

The present thesis is organized into six distinct chapters, each of which fulfills a specific objective in the comprehensive investigation of the research subject matter. Chapter 1 serves as the introductory section of this thesis, wherein the underlying motivation driving the study is established. Furthermore, this chapter highlights the significant contributions made by the research and provides a comprehensive outline of the overall structure of the thesis. Chapter 2 of this thesis aims to establish a comprehensive understanding of fundamental concepts that are crucial to the subject matter. These concepts include blockchain, transactions, blocks, wallets, Shamir's secret sharing, distributed key generation, smart contracts, executor for decentralized applications (DApps), and social login for Web3. By delving into these concepts, this chapter lays the groundwork for the subsequent analysis and exploration of the topic at hand. Chapter 3 of this study presents the proposed solution, which outlines an innovative approach that has been adopted to effectively tackle the challenges that have been identified. Chapter 4 delves into an in-depth analysis of the technical issues and design considerations encountered throughout the implementation process. This chapter aims to address and shed light on the various challenges that were confronted during the execution of the project. Chapter 5 of this study presents a comprehensive evaluation of the proposed solution, offering valuable insights into its performance and usability. In conclusion, Chapter 6 serves as the final segment of this thesis, wherein the findings are succinctly summarized and potential avenues for future research are deliberated upon. The inclusion of a reference section within a thesis serves the purpose of meticulously documenting all the sources that have been cited throughout the research, thereby upholding the academic integrity of the study.

CHAPTER 2. BACKGROUND

The background section of this thesis provides a comprehensive overview of the key concepts and technologies that serve as the foundation for our proposed solution. This chapter examines the fundamental characteristics of blockchain technology, what social login for Web3 is, what a smart contract is, and the fundamental comprehension and utilization scenarios of Shamir's secret sharing and distributed key generation. Understanding these concepts is crucial for appreciating our solution's motivations and its potential impact on the decentralized digital landscape.

2.1 Blockchain

Blockchain technology has emerged as a revolutionary innovation with the potential to transform industries and revolutionize how digital transactions are conducted. It provides a decentralized and transparent platform for secure and unchangeable record-keeping, eliminating the need for intermediaries and facilitating peer-to-peer interactions. This chapter provides a concise introduction to blockchain technology, highlighting its historical context, the problem it seeks to solve, and the primary contributions of its first author. In 2008, an anonymous person or group of people using the alias Satoshi Nakamoto [3] introduced the concept of blockchain for the first time. The seminal whitepaper titled "Bitcoin: A Peer-to-Peer Electronic Cash System" by Satoshi Nakamoto outlined the fundamental principles and architecture of blockchain technology as a solution to the issues of trust and decentralized digital currency. Bitcoin's introduction of blockchain represented a significant milestone in the evolution of cryptocurrencies and decentralized systems. Traditional centralized systems' lack of trust and security is the issue blockchain seeks to address. The reliance of centralized systems on a single trusted authority to validate and authenticate transactions leaves room for manipulation, deception, and censorship. Blockchain technology addresses these issues by establishing a decentralized network of nodes where consensus mechanisms guarantee the validity and integrity of transactions without requiring a central authority. The first author, Satoshi Nakamoto, introduced a secure and decentralized framework for digital currency transactions, laying the groundwork for blockchain technology. Combining existing cryptographic techniques, such as hash functions and digital signatures, with a distributed ledger system was Nakamoto's most significant innovation. This innovation facilitated the creation of a transparent and tamper-resistant ledger of transactions, ensuring the integrity and immutability of blockchain data. Since Nakamoto's original work, blockchain technology has expanded beyond cryptocurrencies such as Bitcoin. It has implications in numerous industries, including finance, supply chain management, and healthcare, among others. The blockchain's decentralized nature provides opportunities for greater transparency, efficiency, and trust in these industries, paving the way for innovative solutions and new business models.

2.1.1 Transactions

"Transactions are the most important part of the Bitcoin system. Everything else in bitcoin is designed to ensure that transactions can be created, propagated on the network, validated, and finally added to the global ledger of transactions (the blockchain). Transactions are data structures that encode the transfer of value between participants in the Bitcoin system. Each transaction is a public entry in bitcoin's blockchain, the global double-entry bookkeeping ledger" according to "Mastering Bitcoin: Unlocking Digital Cryptocurrencies" by Andreas M. Antonopoulos [5], which implies that transactions are fundamental components of blockchain technology, serving as the building blocks for the transfer and exchange of digital assets. Blockchain networks' security and trustworthiness rely heavily on transactions. They are intended to be verifiable and immutable, providing a transparent and auditable log of all blockchain activities. By recording each transaction on the distributed ledger, participants are able to trace the history and origin of digital assets, fostering accountability and preventing double spending. Multiple stages are involved in the creation of a transaction. The sender initiates the transaction by specifying the recipient's address and the desired transfer amount. The originator then signs the transaction with their private key, ensuring the transaction's authenticity and integrity. Once the transaction has been digitally signed, it is disseminated to the network for validation and inclusion in a block. The validation procedure involves verifying the digital signature of the transaction using the sender's public key, thereby ensuring that the transaction has not been tampered with and that the originator has sufficient funds to complete the transfer. The transaction is submitted to a pool of pending transactions awaiting confirmation after validation. Miners, who are tasked with safeguarding the blockchain, select transactions from the pool and incorporate them into a new block. A consensus mechanism, such as proof-of-work or proof-of-stake, is then used to add the transaction to the blockchain.The creation of transactions on the blockchain enables participants to transmit digital assets without the need for intermediaries in a transparent and secure manner. It assures the system's integrity by employing cryptographic techniques to authenticate and authorize transactions, thereby rendering the process tamper-proof and fraud-resistant.

2.1.2 Blocks

A block in a blockchain is a fundamental element that is crucial to the network's structure and functionality. It functions as a repository for a collection of transactions and other pertinent data. Each block is comprised of a block preamble, which includes metadata such as the block's unique identifier, timestamp, and a reference to the previous block, establishing a chronological order. The block contains transactions, which represent numerous actions within the blockchain network. These transactions include sender and recipient addresses, digital signatures for authentication, and additional pertinent information. The block also contains a Merkle tree root [6], which provides an efficient method for verifying the validity of transactions contained within the block. In addition, each block is allocated a unique block hash that is generated by a cryptographic hash function. This block hash serves as a digital fingerprint for the block's content and ensures its immutability. In proof-of-work consensus algorithms, for instance, miners compete to find a nonce value that, when combined with the block header, satisfies specific criteria, thereby adding a layer of security through the solution of computational puzzles. The block functions as the fundamental unit of the blockchain, enabling secure, transparent, and efficient transaction storage and verification.

2.1.3 Wallet

2.1.3.1 Key and address

Key and address are foundational concepts pertaining to user identification and transaction security in the context of blockchain technology and cryptocurrencies. A key, also known as a cryptographic key, is a fragment of information utilized in cryptographic algorithms for a variety of purposes, including encryption, decryption, and digital signatures. Typically, in the context of blockchain, keys are used to secure access to digital assets and to authenticate transactions. There are various mathematically related key categories, including private and public keys. A private key is a secret, randomly generated number that is kept covert by the user. It is used to generate digital signatures, which verify the integrity and authenticity of transactions. The private key should be stored in a secure location and never shared with anyone. If a third party obtains access to the private key, they may be able to take control of the associated digital assets. On the other hand, an address is a cryptographic representation of a user's public key. In a blockchain network, it is a string of alphanumeric characters that functions as a unique identifier for receiving transactions or messages. The public key is used to generate addresses, but they do not disclose any information about the private key. When sending a transaction to a particular user in a blockchain network, the recipient's address is used as the destination. The address functions as a pseudonymous identifier, providing privacy and security. The recipient can then access and manage the digital assets associated with that address using their private key.

2.1.3.2 Wallet

A cryptocurrency wallet is a software application or hardware device that enables users to store, administer, and interact with their digital assets in a secure manner. Wallets play a crucial role in the adoption and use of cryptocurrencies by both consumers, enhancing the overall experience with a variety of advantages. Wallets provide a convenient and intuitive interface for managing digital assets. They provide a secure solution for storing private keys, which are required for accessing and controlling cryptocurrencies. Wallets enable users to transfer and receive funds, track their transaction history, and monitor their account balances by storing private keys securely. Wallets typically include features such as address book administration, transaction history, and real-time market data, providing users with a comprehensive set of tools for managing their cryptocurrency holdings. One important aspect of wallet security is the implementation of industry standards, such as the BIP39 specification [7], in the generation and management of mnemonic phrases or seed phrases. The BIP39 specification ensures that wallets adhere to a standardized method for generating mnemonic phrases, which are human-readable sets of words. These phrases can be used to derive the cryptographic keys necessary to access and manage cryptocurrency funds. By adopting the BIP39 specification, wallets provide users with a consistent and reliable way to backup and restore their wallets, offering an additional layer of security and ease of use. MetaMask [8] is a prominent example of a cryptocurrency wallet. MetaMask is a wallet extension for web browsers that enables users to interact with Ethereum-based decentralized applications (DApps) directly from the browser. It provides a user-friendly and secure interface for interacting with Ethereum accounts and the blockchain. MetaMask provides a straightforward and intuitive interface that integrates seamlessly with popular web browsers such as Chrome, Firefox, and Brave. Within minutes, users can install the MetaMask extension and configure their Ethereum wallet. After configuring the wallet, users can access their Ethereum accounts, view their token balances, and conduct transactions.

2.1.4 Consensus

The concept of consensus holds paramount importance in the realm of blockchain technology as it serves to guarantee the agreement and validity of transactions throughout the network. This mechanism encompasses a process by which decentralized nodes within the network collectively establish agreement regarding the current state of the blockchain. This paper examines two prevalent consensus algorithms, namely Proof of Work (PoW) and Proof of Stake (PoS).

PoW consensus algorithm, initially pioneered by Bitcoin [3], serves as the foundational mechanism for validating transactions and maintaining the integrity of the blockchain network. In the context of PoW, the participants referred to as miners engage in a competitive process aimed at solving intricate mathematical puzzles. In the realm of blockchain technology, the initial miner who successfully unravels the intricate puzzle is duly acknowledged and bestowed with a reward, subsequently appending a novel block to the existing chain of transactions. The aforementioned process necessitates a substantial amount of computational resources and incurs a considerable level of energy expenditure. The security of a blockchain system is upheld through the utilization of PoW, which effectively deters malicious entities from tampering with previous transactions. This is achieved by imposing a significant computational burden on any attempts to modify the blockchain's historical records.

PoS [4] consensus algorithm serves as a viable alternative to the PoW mechanism, with the primary objective of mitigating the concerns pertaining to energy consumption commonly associated with PoW. In the PoS consensus mechanism, the selection of validators to generate new blocks is determined by their cryptocurrency holdings and their willingness to "stake" said holdings as collateral. The selection of validators is typically conducted through a deterministic procedure, which frequently takes into consideration factors such as the magnitude of their stake and the duration for which they have maintained it. PoS consensus mechanism is widely acknowledged for its superior energy efficiency in comparison to the PoW mechanism, primarily due to its reduced reliance on intensive computational resources.

The utilization of both PoW and PoS consensus mechanisms presents distinct benefits and limitations. PoW consensus mechanism is renowned for its robust security measures, albeit at the cost of significant resource consumption. Conversely, the PoS protocol boasts energy efficiency advantages, yet it may exhibit vulnerability to specific forms of attacks. The selection between PoW and PoS is contingent upon the distinct objectives and prerequisites of a blockchain network.

2.2 Shamir's secret sharing

Shamir's secret sharing is a cryptographic algorithm that divides a secret into multiple shares, which can then be distributed to various participants. The secret can only be reconstructed by combining a substantial number of shares. The algorithm was created in 1979 by Adi Shamir and is based on polynomial interpolation. To implement Shamir's secret sharing, a secret is first selected, and then a polynomial of a specific degree is generated with the secret as the constant term. The polynomial is subsequently evaluated at particular points to generate the shares. Each participant receives a portion of the polynomial curve that corresponds to a specific point. Any subset of shares, so long as it satisfies a certain threshold, can be used to reconstruct the original secret, according to Shamir's secret sharing scheme.
Here is a straightforward illustration of how Shamir's secret sharing works. Suppose we wish to divide a secret value of 42 into 5 portions with a threshold of 3. By evaluating a polynomial at various points, the shares are generated. Share 1: (1, 17), Share 2: (2, 23), Share 3: (3, 38), Share 4: (4, 14) Share 5: (5, 7) x represents the point on the polynomial curve, while y is the value of the polynomial at that point. Now, we need at least three shares to reconstruct the secret. Consider the shares 2, 3, and 4. We can use these shares to interpolate the polynomial and determine the value at x = 0 that corresponds to our confidential value of 42 by employing interpolation. Using the Lagrange interpolation formula [9], the secret can be calculated:

$$\text{Secret} = \frac{23 \cdot (0-3) \cdot (0-4)}{(2-3) \cdot (2-4)} + \frac{38 \cdot (0-2) \cdot (0-4)}{(3-2) \cdot (3-4)} + \frac{14 \cdot (0-2) \cdot (0-3)}{(4-2) \cdot (4-3)}$$

After simplifying the equation, the hidden value is determined to be 42.

2.3 Distributed Key Generation

The Distributed Key Generation (DKG) protocol is a cryptographic mechanism that allows multiple parties to generate a shared secret key collaboratively without relying on a single trusted authority. It ensures that no one party has complete knowledge of the secret key, thereby enhancing security and decreasing the likelihood of a single point of failure. In a DKG protocol, the participating parties collaborate to generate and distribute portions of the secret key. Combining these shares mathematically yields the final confidential key. Protocol phases include key generation, distribution, verification, and reconstruction. The primary benefits of DKG protocols are their security and resilience. The protocol mitigates the risk of a single party compromising the confidential key by distributing the key generation process across multiple parties. Even if some parties are compromised, the final key will remain secure if a minimum number of trustworthy parties are involved. Protocols for Distributed Key Generation have applications in numerous disciplines, including secure multi-party computation, cryptographic key management, threshold cryptography, and secure communication protocols. They provide a robust mechanism for establishing shared secret keys in situations where there is little or no trust between participants.
The Pedersen DKG protocol, proposed by Torben Pedersen [10] in 1991, was used in the thesis. The Pedersen DKG protocol utilizes polynomial interpolation techniques and cryptographic primitives to achieve secure and distributed key generation. It provides a robust mechanism for establishing shared secret keys without relying on a trusted central authority. The protocol involves several steps, including key generation, sharing, verification, and reconstruction.The Pedersen DKG protocol utilizes polynomial interpolation techniques and cryptographic primitives to achieve secure and distributed key generation. It provides a robust mechanism for establishing shared secret keys without relying on a trusted central authority. The protocol involves several steps, including key generation, sharing, verification, and reconstruction.

2.4 Smart contract

2.4.1 History and defination

Smart contracts are agreements that automatically carry out their obligations because they are encoded in code. By eliminating the need for middlemen and supplying a safe and decentralized method to facilitate and enforce agreements or transactions, these contracts automatically execute and enforce themselves. The idea of smart contracts has been around since the 1990s, and computer scientist Nick Szabo [11] is credited with coining the term. However, smart contracts did not receive much attention or widespread use until the advent of blockchain technology, particularly with the launch of Ethereum [4] in 2015. A Turing-complete programming language was introduced by Ethereum, a decentralized blockchain platform, allowing for the creation and execution of sophisticated smart contracts. This innovation paved the way for the development of decentralized applications (DApps) that might use smart contracts to secure and automate a variety of activities, including voting systems, supply chain management, and financial transactions. Since then, smart contracts have become more well-known and are being investigated in a variety of sectors and industries for their potential to transform conventional corporate operations. Their immutability and transparency, along with the ability to automate processes and get rid of middlemen, have the potential to improve workflow, boost productivity, and cut costs.

2.4.2 Practical use cases

Smart contracts have become a game-changing technology with several real-world applications in a wide range of industries. These self-executing contracts, which are inscribed on a blockchain, allow for secure and automated transactions, doing away with the need for middlemen and enhancing participant trust. Smart contracts have transformed lending platforms, decentralized exchanges, and yield farming protocols in the field of DeFi [12]. Smart contracts offer a transparent and effective ecosystem for decentralized financial applications by automating financial transactions and following established regulations. Likewise, supply chain management has benefited from smart contracts. By facilitating seamless tracking and verification of commodities along the supply chain, these contracts improve traceability, lower fraud, and streamline logistical operations. Smart contracts have simplified real estate transactions by enabling property transfers, escrow services, and rental agreements. Smart contracts increase efficiency and transparency in the real estate sector by doing away with middlemen and automating repetitive operations.

In this thesis, the smart contract plays a pivotal role in the ecosystem by fulfilling a multitude of significant responsibilities. The storage and maintenance of configuration updates for the Perdesen Distributed Key Generation (DKG) protocol is a primary responsibility. The smart contract is responsible for managing a whitelist of decentralized applications (DApps) that utilize the aforementioned solution. This mechanism ensures that only authorized DApps are able to participate in the protocol. The smart contract is designed to enable the asynchronous execution of the Perdesen DKG protocol rounds, which is a fundamental feature of its functionality. The process entails the antecedent creation of cryptographic keys through the secure retention of encrypted shares for each node involved in the operation. The implementation of a smart contract facilitates the asynchronous execution of the key generation procedure, thereby enhancing operational efficiency and scalability. The transparent management of the key generation process is regarded as a fundamental characteristic of the smart contract. The implementation of transparency in the process is paramount to guaranteeing the privacy and security of participants' private keys. The provision of transparency enables participants to authenticate the advancement and soundness of the key generation procedure while upholding the confidentiality of their private key data. The smart contract assumes a crucial function in the verification of signatures from nodes that are involved in the process. The implementation of a verification process within the Perdesen Distributed Key Generation (DKG) protocol serves to guarantee that exclusively legitimate users are allocated roles during each round of the cryptographic scheme. Through the process of signature verification, the smart contract ensures the integrity and authenticity of the nodes involved, thereby augmenting the overall security and dependability of the protocol.

2.5 Executor for DApps

An essential part of enabling the execution of transactions on the blockchain network is played by the executor of a DApp. The backend element tasked with connecting with the blockchain and carrying out transactions on behalf of users is referred to as the executor in the context of DApps. The user's private key or mnemonic being kept in the backend is one typical method for carrying out transactions in a DApp. The private key, which is used to sign transactions and verify identities, is represented by a series of phrases known as the mnemonic. The executor can access the mnemonic when necessary to sign transactions on behalf of the frontend by securely keeping it in the backend. The executors play a specific role in the thesis as the verifiers and assigners of a Perdesen DKG protocol round within the smart contract. The decentralized system's executors serve as dependable parties and are in charge of assuring the fairness and security of the round assignment procedure.

2.6 Social login for Web3

2.6.1 Web3 and Dapps

Web3 and Decentralized Applications (DApps) have emerged as critical components of the evolution of the internet and digital ecosystems. Web3 is the vision of a more decentralized and user-centric internet, in which individuals have greater control over their data, identity, and digital interactions. It is a collection of technologies, protocols, and frameworks designed to empower users, cultivate trust, and facilitate peer-to-peer interactions. DApps, on the other hand, are applications that are created on top of decentralized networks and typically utilize blockchain technology. These applications inherit Web3's fundamental principles, including decentralization, transparency, and user ownership. From financial services and governance platforms to gaming and social media applications, they provide a variety of features. The development and adoption of Web3 and DApps have been supported by a growing body of research and innovation. Several academic papers and technical publications have contributed to the advancement of these technologies. For instance, the paper by Wood et al. titled "Ethereum: A Secure Decentralized Generalized Transaction Ledger" provides a comprehensive overview of the Ethereum platform and its underlying principles [4]. Another significant contribution is the work by Swan, who explores the concept of "Token Economy" in his book "Token Economy: How the Web3 Reinvents Value Exchange." The book delves into the transformative potential of tokenization and its implications for various industries [13]. Moreover, the paper by Buterin et al. titled "A Next-Generation Smart Contract and Decentralized Application Platform" introduces the Ethereum platform, highlighting its unique features and use cases [14]. This paper serves as a foundational reference for understanding the capabilities and potential of DApps built on Ethereum.

2.6.2 Social login and OAuth

Social login is a prevalent method of authentication that enables users to log in to websites and applications using their existing social media accounts. Users are no longer required to establish new accounts and remember additional login credentials. Instead, users can merely click on a social media button, such as "Sign in with Facebook" or "Sign in with Google," to authenticate themselves. Social registration is supported by the OAuth2 (Open Authorization 2.0) [15] protocol, which provides a secure and standardized authentication framework. When a user logs in with a social media account, the website or application sends them to the respective social media platform for authentication. The user is then presented with a consent interface that describes the data to which the website or application requests access. Once the user grants permission, the social media platform provides the website or application with an access token that can be used to retrieve user information and authenticate the user's identity. Using OAuth2 for social authentication has multiple advantages. It increases security by removing the need for websites and applications to store user credentials. Instead, the obligation for authentication falls on the shoulders of the most reputable social media platforms. Second, social login streamlines the user experience by allowing users to log in with a few clicks and avoid the inconvenience of creating new accounts. It also allows websites and applications to utilize the extensive user profile data available on social media platforms, including user names, profile pictures, and email addresses, for personalization and customization.

2.6.3 Social login benefits DApps

Web3's technical complexity is one of the most significant obstacles it presents to normal consumers. The underlying technologies, such as blockchain, cryptographic keys, and smart contracts, can be complex and challenging for non-technical individuals to comprehend. Comprehending and traversing these

complexities can be an impediment to entry, impeding widespread adoption and utility. In addition, the decentralized nature of Web3 platforms may result in fragmented user experiences and inconsistent user interfaces, making it difficult for non-technical users to effectively navigate and interact with decentralized applications. Simplifying the user experience and enhancing the accessibility of Web3 technologies will be crucial for overcoming these obstacles and ensuring that regular users can reap the full benefits of Web3.
This thesis proposes a solution to improve the usability of Web2 [16] applications by integrating smart contracts, the Pedersen Distributed Key Generation (DKG) protocol, and Shamir secret sharing. The present study endeavors to propose a solution that seeks to mitigate the difficulties encountered by laypersons in comprehending and maneuvering the intricate technicalities of Web3. Additionally, the solution endeavors to guarantee the secure generation and distribution of cryptographic keys, as well as the safeguarding of data.

CHAPTER 3. SOLUTION

3.1 System's characteristics

This study examines the efficacy and adaptability of a system that utilizes conventional social login methods to augment the UX of DApps. The following characteristics can be more detailed:

Security - combining the potential benefits of blockchain smart-contract, SSS, and Pedersen DKG protocol. The security infrastructure of the system is established upon the utilization of blockchain smart contracts. Programmable contracts, operating on a decentralized network, guarantee the attributes of transparency, immutability, and tamper-proof execution of operations. Shamir secret sharing is used to fortify the protection of cryptographic keys and other forms of private information. This method entails breaking up a secret into smaller pieces and giving them to several parties. The Petersen DKG protocol distributes and secures cryptographic key generation, ensuring system security. This system lets a group of people generate a shared cryptographic key without anyone having the full key.

Efficacy and adaptability - By leveraging blockchain's decentralized nature, Web3Auth eliminates the reliance on centralized identity providers, reducing the potential for single points of failure and enhancing security. Additionally, blockchain-based identity systems enable instant verification of user credentials, eliminating the need for lengthy verification processes and reducing transaction times. The open solution's architecture enables seamless integration with any DApps, making it easier for developers to adopt and implement with their products.

User-friendly - This solution supports popular authentication mechanisms such as social login which are widely used in the traditional web. This familiarity allows users to leverage their existing accounts and authentication methods, reducing the learning curve and providing a seamless transition into the Web3 space. Users can easily authenticate their identities across multiple DApps using a unified and standardized protocol, without the need to remember and manage multiple usernames and passwords. This eliminates the hassle of creating and maintaining numerous accounts, making the user experience more streamlined and efficient.

Scalability - Compatible with any type of social authentication, including Facebook, Google, and Twitter. Utilizing a decentralized architecture increased the request's performance and volume by spreading it across multiple executors and parallel handling processes.

3.2 Overall of system

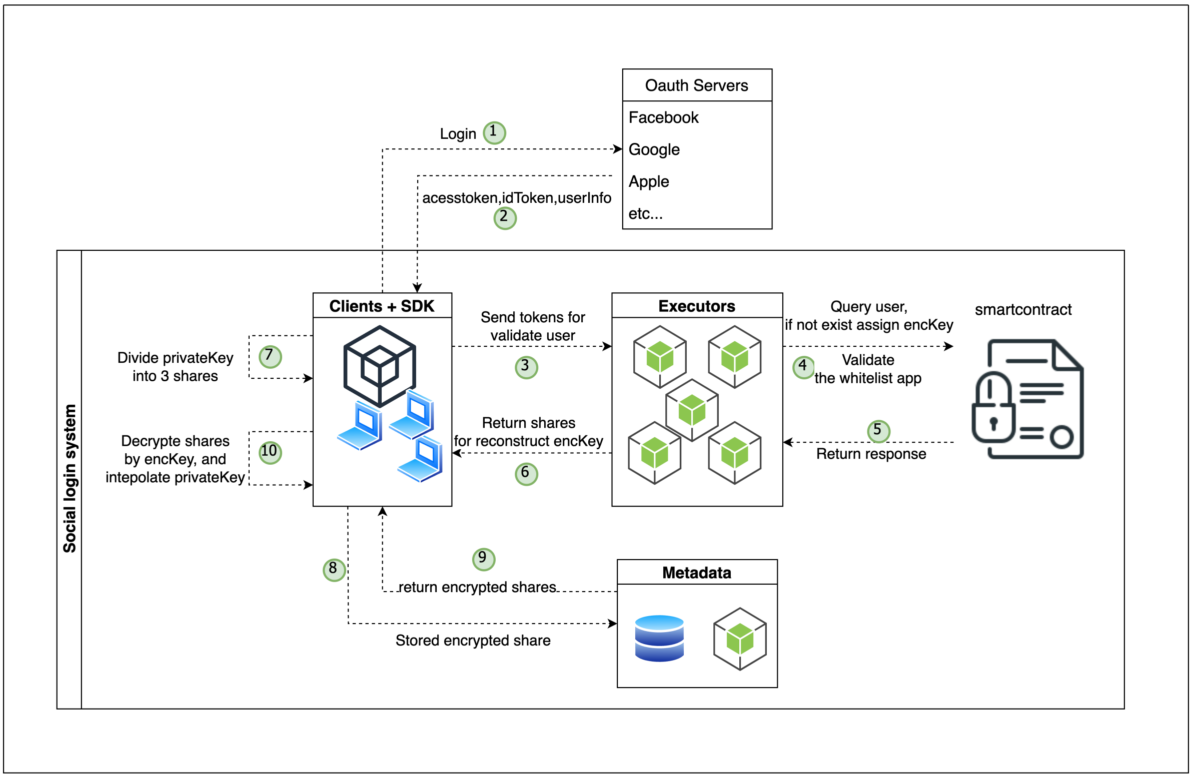

Figure 3.1: Overall the system

Figure 3.1 depicts the overall processing of a request from a user utilizing the social login system. Because the process is more complex, it has been divided into three sub-processes for clarity.

3.2.1 Auth0 connection

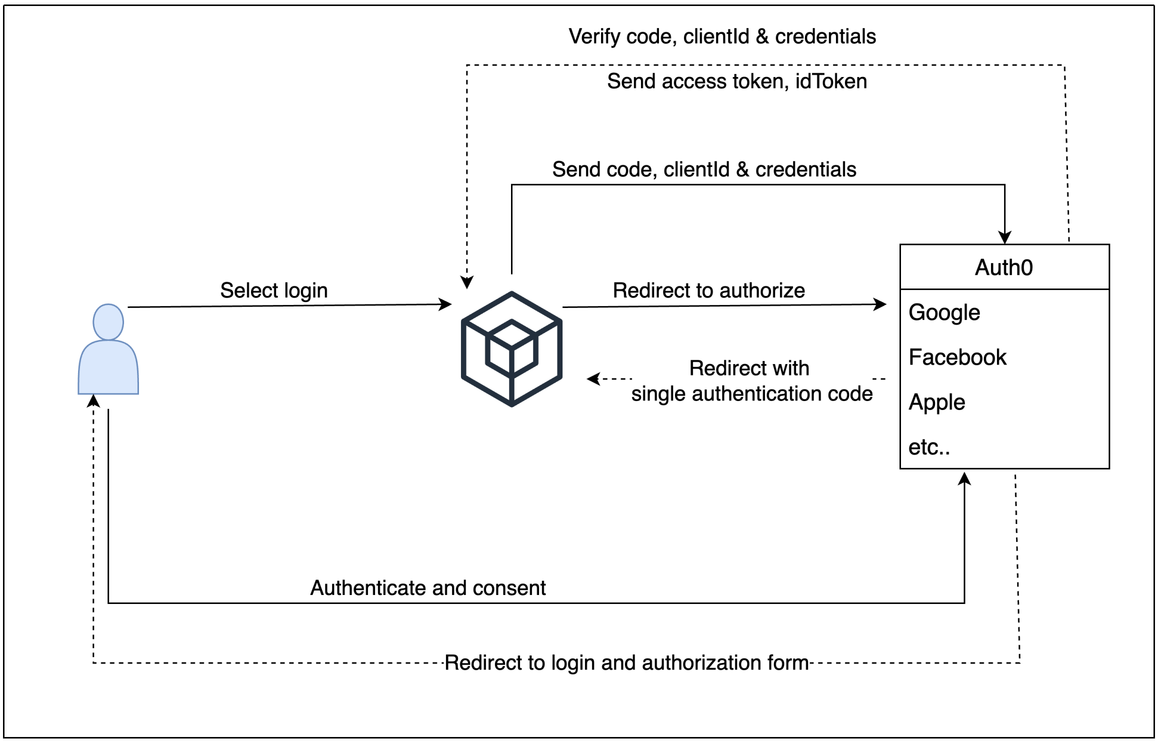

Figure 3.2: Auth0 connection

Figure 3.2 is a graphical representation of the Auth0 authentication and authorization solution. When the user selects the intended authentication type within the application, the process begins. This could be accomplished using a traditional username and password or other methods, such as social registration with Google or Facebook. After selecting the authentication type, the application redirects the user to the Auth0 server's authorization endpoint. The user is then presented with a login and authorization prompt, where they can securely authenticate themselves and grant any necessary permissions. This step ensures that only authorized users have access to the application's resources and can conduct particular actions. The Auth0 server redirects the user back to the application along with an authorization code following successful authentication and assent. The application then exchanges this authorization code, along with its own credentials and client ID, with the Auth0 server's token endpoint. The Auth0 server verifies the supplied information, including the authorization code, credentials, and client ID, with extreme precision. If the verification procedure is effective, the server issues both an access token and an ID token to the application. Different purposes are served by these identifiers during the authentication and authorization process. The application uses the access token to authenticate subsequent inquiries on the user's behalf. It serves as evidence of authorization and grants access to protected resources or API endpoints. This token is typically included in API request parameters, allowing for secure and seamless communication between the application and server. The ID token, on the other hand, contains vital user information that can be used for identification and authentication. It may contain information such as the user's email address, identity, and other pertinent characteristics. The ID token enables the application to recognize the user and tailor the user's experience within the application. Auth0's extensive features make it a robust and dependable authentication and authorization solution. By leveraging secure protocols and industry best practices, Auth0 assures the privacy and integrity of user data through a secure and efficient process.

3.2.2 Requesting encKey for Executors

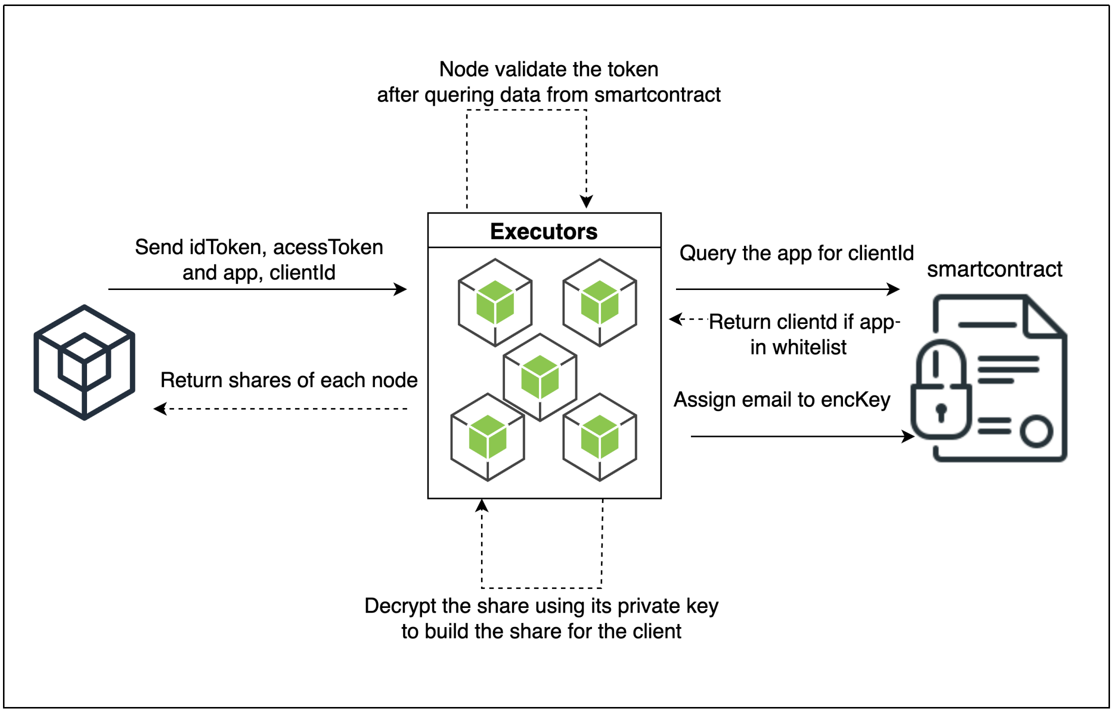

Figure 3.3: Request assign the share

Multiple stages, described in Figure 3.3, are required to ensure secure and effective authentication and authorization. The SDK functions as the interface between the user's device and the system. The SDK sends the idToken or accessToken, as well as the appName and clientId, to the executors when the user initiates the authentication procedure. The executors play a vital role in authenticating the user's credentials and interacting with the smart contract in a secure manner. They begin by querying the smart contract to retrieve pertinent information regarding the accessed app. This information contains the essential authentication and authorization parameters and configuration data. The executors commence the validation procedure upon receiving the data from the smart contract. They validate the integrity and authenticity of the data extracted from the idToken or accessToken. This validation ensures that the user's credentials have not been altered during transmission and are legitimate. If the user attempting authentication is a new user, the executors generate a unique key for them. This key is securely associated with the user's authenticated email address and functions as a digital identifier. This phase ensures that each user within the system has a unique identifier. The executors then receive the shares encrypted by the smart contract in a secure manner. These shares are encrypted with the private keys of the nodes that participated in the distributed key generation procedure. The executors can decrypt the shares using their own private keys, ensuring that the process remains secure and confidential. After decrypting the shares, the executors reconstruct the final share and return it to the SDK. These shares are used as part of the authentication and authorization process, allowing the user to access and interact with decentralized applications (DApps) within the ecosystem in a secure manner. By adhering to this exhaustive and intricate procedure, the system ensures the authentication and authorization of users is efficient and secure. It leverages the capabilities of smart contracts, Shamir secret sharing, and the Petersen DKG protocol to provide a robust and dependable Web3 authentication solution.

3.2.3 Generating and Storing Shares in Metadata

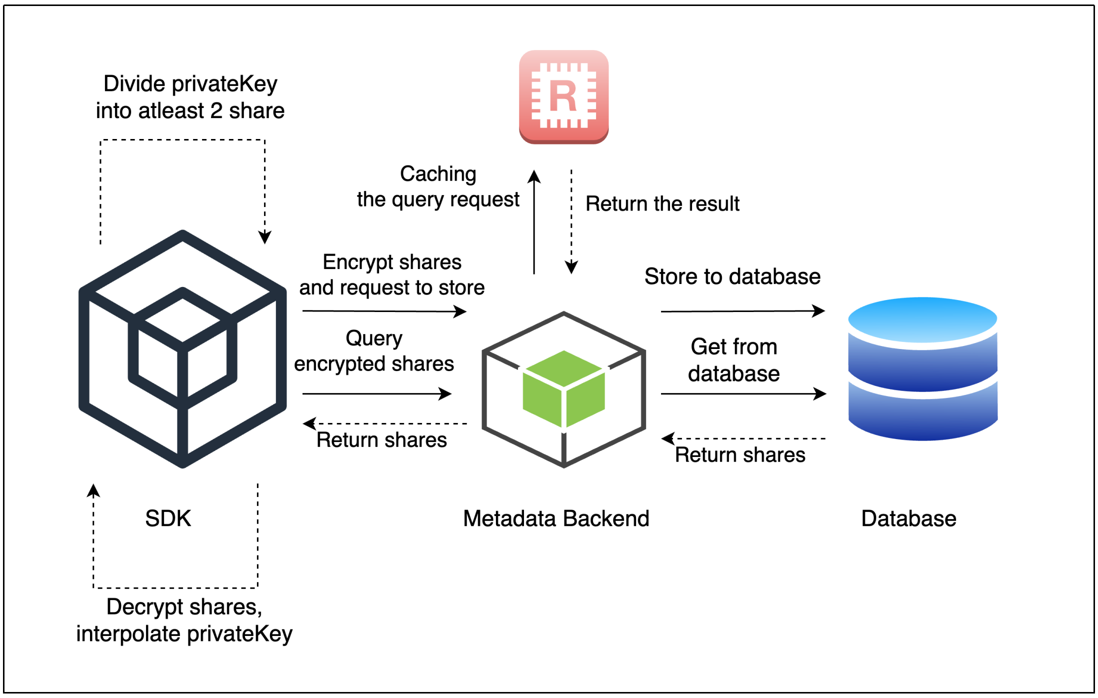

Figure 3.4: Generate Reconstruct shares

Reconstructing the private key in our system, as depicted in Figure 3.4, requires several phases to assure security and dependability. Beginning with Shamir's secret sharing scheme, the SDK divides the private key into at least two portions. Each share is encrypted to ensure its integrity and confidentiality. The encrypted shares are then transmitted to the metadata backend, where they are housed securely in the database. The SDK initiates a query request to the metadata repository when the need to reconstruct the private key arises. The backend retrieves from the database the encrypted shares associated with the specified user and application. The query result, including the encrypted shares, is cached in a Redis database to improve performance, enabling quicker retrieval for subsequent requests. After receiving the encrypted shares, the SDK initiates the reconstruction process. Each share is decrypted by the SDK using the corresponding decryption keys. These decryption keys are managed securely within the SDK, and only authorized processes have access to them. The SDK recovers the original confidential information contained within each share by decrypting the shares. The SDK combines the decrypted shares using interpolation algorithms, such as Lagrange interpolation, to generate the final private key. This method entails mathematical calculations that interpolate the missing portions of the private key using the available shares. The SDK effectively reconstructs the original private key through interpolation. Throughout this process, strong encryption algorithms and secure key management procedures guarantee the privacy and integrity of the private key. The use of Shamir's secret sharing scheme increases security by distributing the key across multiple shares, making it resistant to single points of failure or compromise. By implementing this comprehensive and secure reconstruction procedure, our system ensures the safe and dependable retrieval of the private key, allowing users to securely access their Web3 ecosystem accounts and conduct authorized actions.

3.3 Asynchronous create encKey process

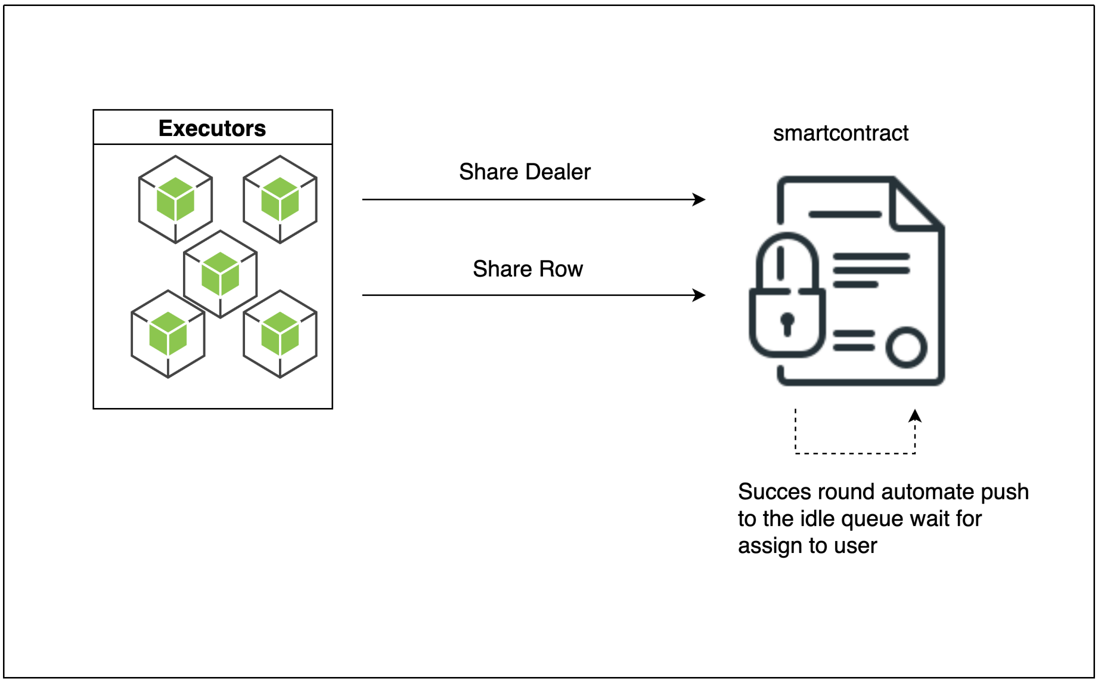

Figure 3.5: Asynchronous create encKey

The asynchronous creation of the encKey process is a crucial component of the architecture of the entire system, which is intended to maximize efficiency and resource utilization. By pre-generating a pool of idle encKeys, the system ensures that users can be assigned keys swiftly and seamlessly, without running the Distributed Key Generation (DKG) protocol each time. Executors, who are responsible for administering the encKey generation process, initiate the procedure. They continuously monitor the status of the current round and the number of inactive keys. When the system detects a need for new keys, such as when the present state is null or in the waitfordealer phase, and the number of idle keys falls below the expected threshold, the executors initiate the next round. During the initial phase of the round, known as the "sharing dealer" phase, the executors pool their individual confidential shares and share them with the designated sharing dealer. This phase ensures that each executor contributes without divulging sensitive information to the overall secret key. By collaborating, the executors distribute the secret shares in a manner that preserves the keys' security and integrity. Once the sharing dealer phase is successfully concluded, the protocol advances to the next phase, known as "waitforrows." In this phase, the executors use the shared secrets obtained in the preceding phase to generate their own public key shares. These public key shares are essential to the completion of the key generation process, as they assure the authenticity and validity of the user-assigned keys. The executors are well-prepared for the assignment of encKeys to users following the conclusion of the "waitforrows" phase. When a user requests their key, the system seamlessly maps the correct round to their designated key and provides it. This asynchronous approach to encKey creation improves the system's overall performance by decreasing the time and computational resources necessary for key assignment. The system accomplishes a seamless and secure workflow for encKey generation and assignment by incorporating the power of the DKG protocol and the efficiency of the asynchronous creation process. This method not only assures efficient resource utilization, but also improves the system's scalability and performance, allowing it to effectively handle a large number of user requests.

4.1 Requirement analysis

4.1.1 General usecases diagram

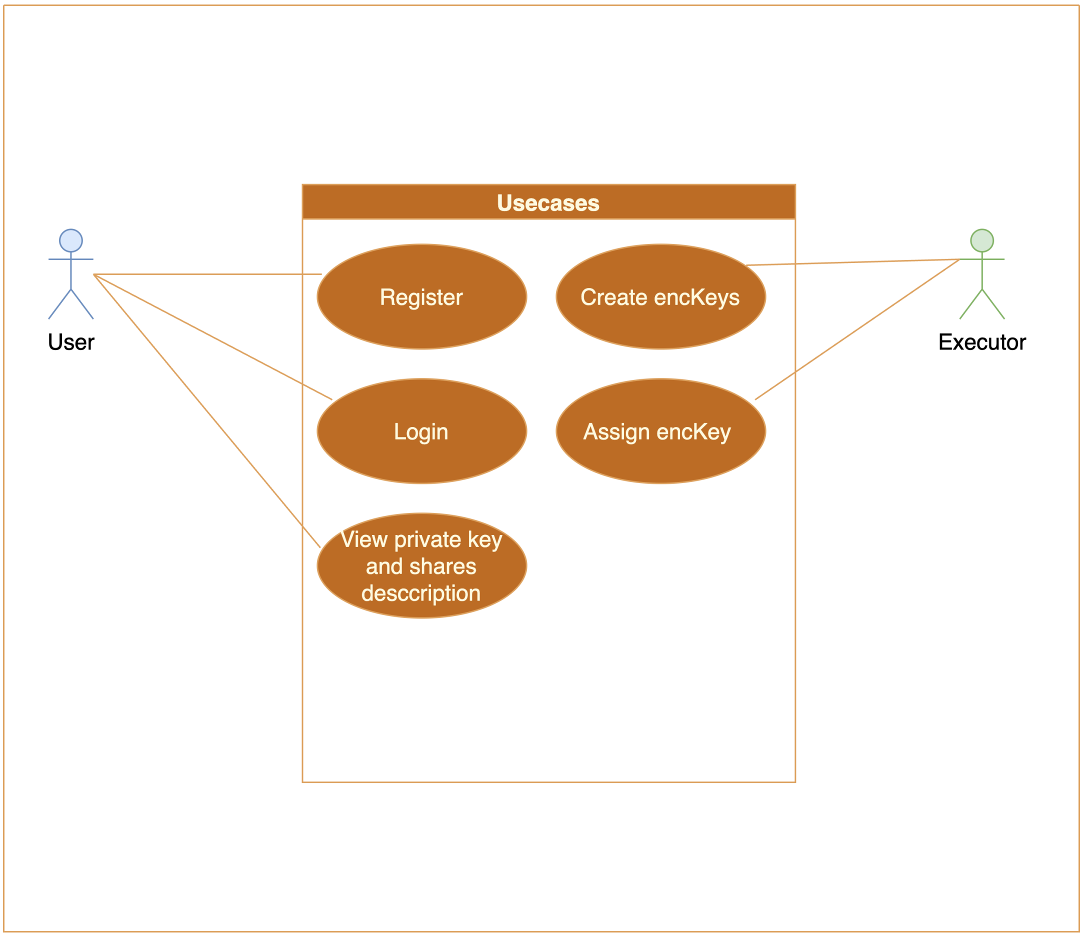

Figure 4.1: General usecases diagram

The figure 4.1 depicts the general usecases of Social login system. There are two actors included in the system. The first one is User who will use the main function social login. The table below describes more detail about functionalities of usecase:

No	Use Case	Description
1	Regiser	User using their social account such as: Google, facebook, ... for signing up the web3 spaces
2	Login	User using their social account such as: Google, facebook, ... for signing in the web3 spaces
3	View the private key and shares description	User fully controls and manages their private key and their shares.

Table 4.1: User's usecase description

The second participant is the Executor. They are special users with a crucial function who run the system's create encKey and assign encKey processes in the background.

No	Use Case	Description
1	Create encKey	Executor run the phases of Pedersen Protocol for pre-creating the encKey for user to encrypt their share.
2	Provide signature	Executors communicate with one another via signatures; if the number of valid signatures exceeds the threshold, anyone can delegate a key to a specific user via a multi-signature mechanism.

Table 4.2: Executor's usecase description

4.1.2 Usecase specifications and activity diagrams

For a better understanding of the use cases, I'd like to include the corresponding activity diagrams beneath each specification.

Usecase code	UC001	**Usecase name**	Social login
Actor	User		
Pre-condition	At least have 1 social account		
Main flow of event	**No**	**Actor**	**Action**
	1	User	Select social login type
	2	System	Redirect to authorize
	3	Auth0	Redirect to login and authorization form
	4	User	Authenticate and consent
	5	Auth0	Redirect with single use authorization code
	6	System	Send code clientId and credentials to get oauth token
	7	Auth0	Verify code, cliendId, credentials
	8	Auth0	Send idToken, access token and refresh token
	9	System	Authenticate the idToken or access token
	10	System	Assign encKey for user
	11	System	Send shares
	12	System	Contruct the encKey
	13	System	Construct the private Key
Alternative flow of event	**No**	**Actor**	**Action**
	5b	Auth0	Return error if the authenticate and consent fail or rejected
	8b	Auth0	Response with error because fail in verification

Table 4.3: Sign up specification

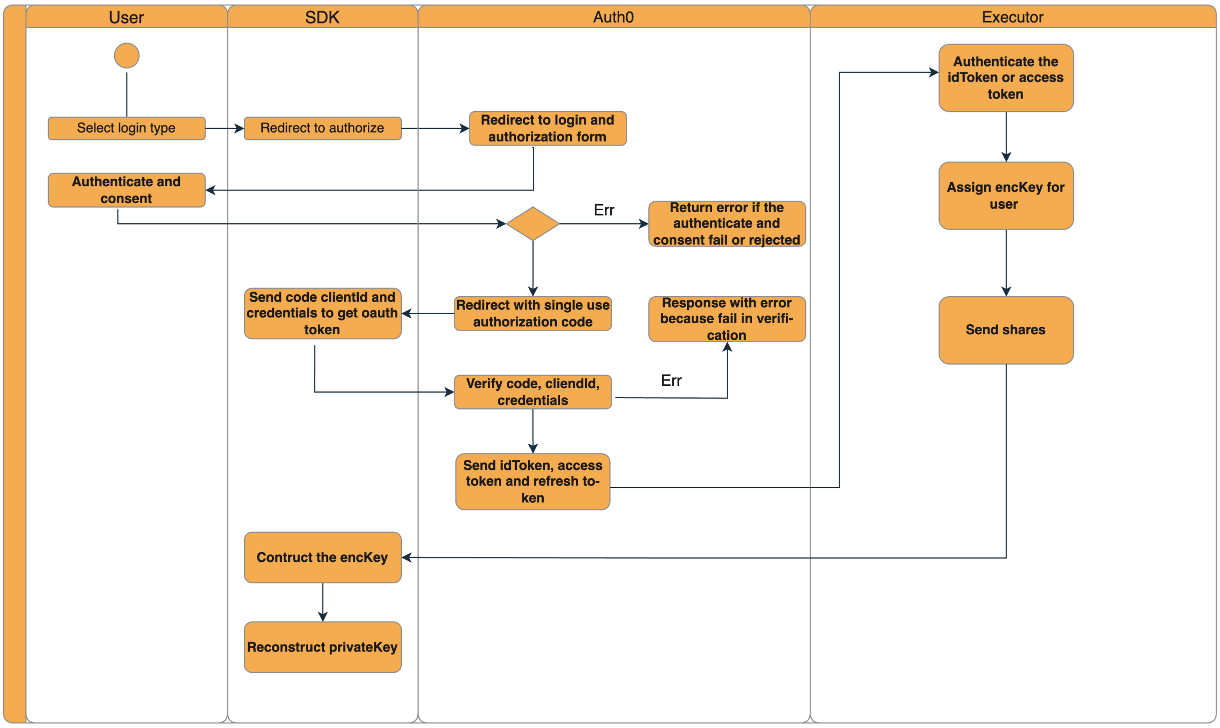

Figure 4.2: Signing up activity

Figure 4.2 is an activity diagram that illustrates the process, how users interact with the system, and the connections between the components. The SDK acts as a bridge between users and the remainder of the system, while Executors and Auth0 validate users' identities. In addition, executors assign and provide the requirements for user encKey and private key construction.

Usecase code	UC001	**Usecase name**	Social login	
Actor	User			
Pre-condition	At least have 1 social account			
Main flow of event	**No**	**Actor**	**Action**	
	1	User	Select social login type	
	2	System	Redirect to authorize	
	3	Auth0	Redirect to login and authorization form	
	4	User	Authenticate and consent	
	5	Auth0	Redirect with single use authorization code	
	6	System	Send code clientId and credentials to get oauth token	
	7	Auth0	Verify code, cliendId, credentials	
	8	Auth0	Send idToken, access token and refresh token	
	9	System	Authenticate the idToken or access token	
	10	System	Send shares	
	11	System	Contruct the encKey	
	12	System	Construct the private Key	
Alternative flow of event	**No**	**Actor**	**Action**	
	5b	Auth0	Return error if the authenticate and consent fail or rejected	
	8b	Auth0	Response with error because fail in verification	

Table 4.4: Sign in specification

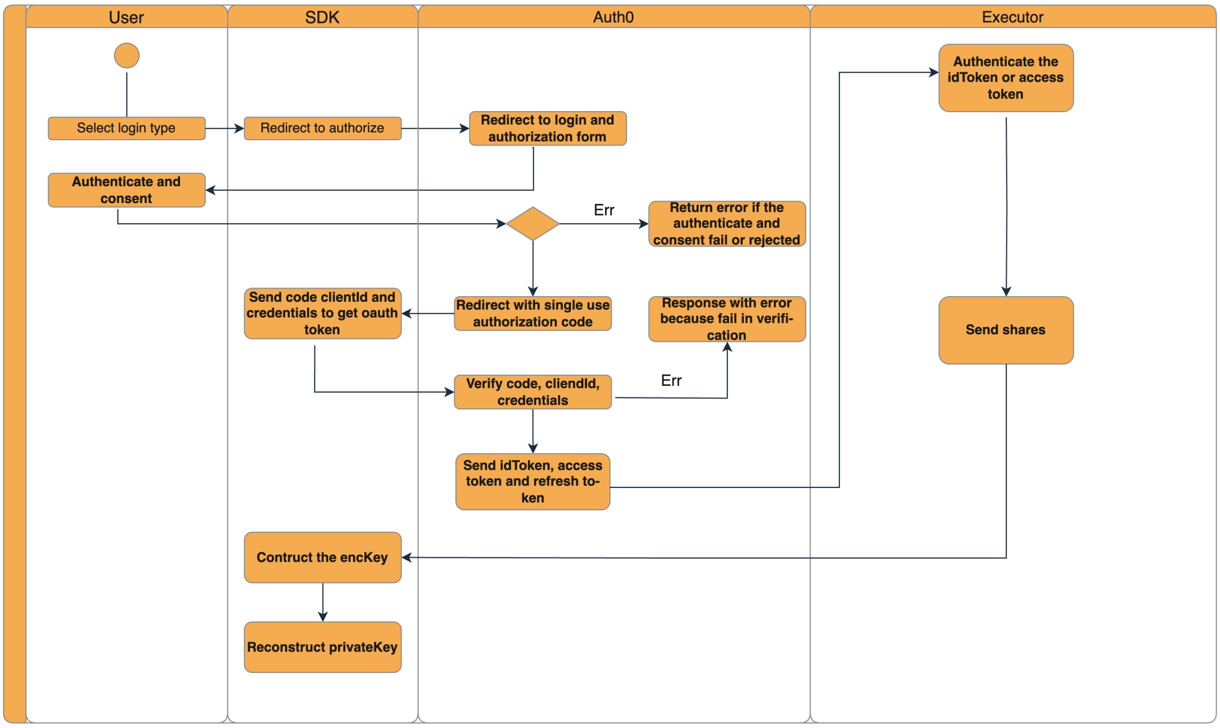

Figure 4.3: Signing in activity

Similar to figure 4.2, figure 4.3 describes in detail how users login into the system and how the system's components interact.

Usecase code	UC001	**Usecase name**	Social login
Actor	User		
Pre-condition	At least have 1 social account		
Main flow of event	**No**	**Actor**	**Action**
	1	User	Select social login type
	2	System	Redirect to authorize
	3	Auth0	Redirect to login and authorization form
	4	User	Authenticate and consent
	5	Auth0	Redirect with single use authorization code
	6	System	Send code clientId and credentials to get oauth token
	7	Auth0	Verify code, cliendId, credentials
	8	Auth0	Send idToken, access token and refresh token
	9	System	Authenticate the idToken or access token
	10	System	Send shares
	12	System	Contruct the encKey
	13	System	Query metadata
	14	System	Return shares description
	15	System	View private key and share description
Alternative flow of event	**No**	**Actor**	**Action**
	5b	Auth0	Return error if the authenticate and consent fail or rejected
	8b	Auth0	Response with error because fail in verification

Table 4.5: View private key and shares description

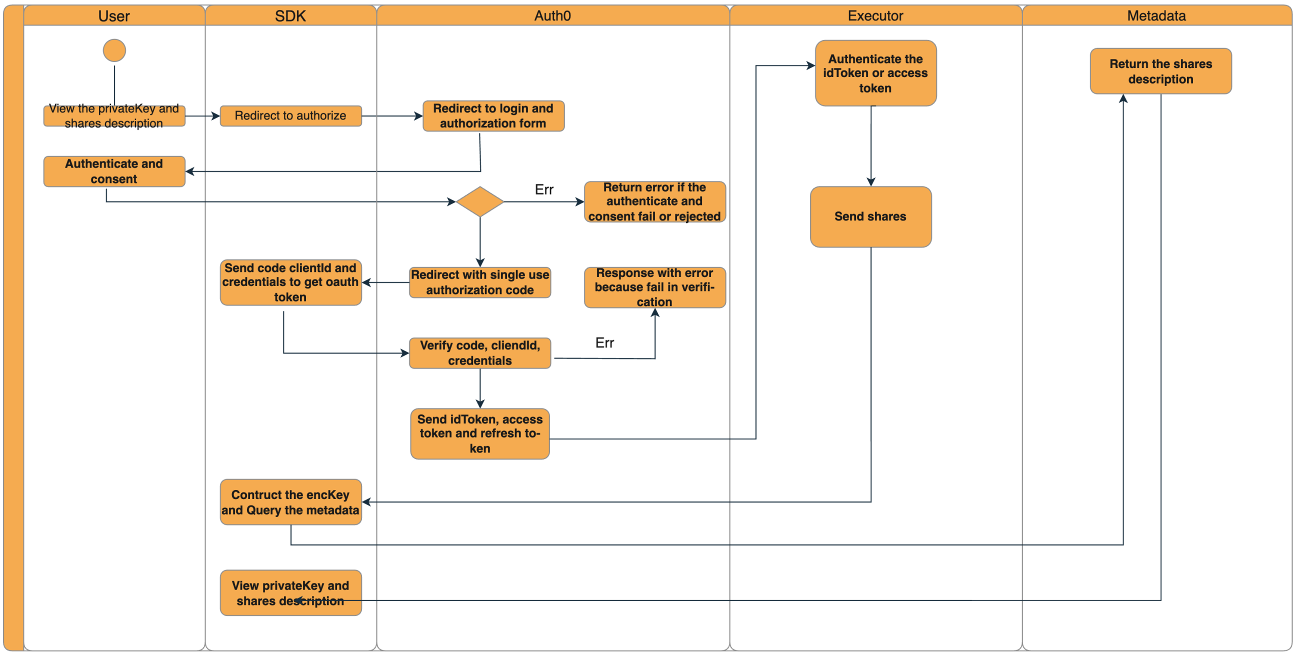

Figure 4.4: View private key and shares description activity

The last use case, showing in 4.4, for the user involves viewing their private key and their shares. This functionality allows users to have full control and management over their shares, even outside the system. By providing access to their private key and the corresponding shares, users can securely store this information in a location of their choice, such as a hardware wallet or an encrypted offline storage. This feature enhances the security and flexibility of the system, as users have the freedom to manage their shares independently and ensure the safety of their cryptographic assets. Additionally, by being able to view their private key and shares, users can also perform offline verifications and audits, ensuring the integrity and accuracy of their cryptographic operations.

Usecase code	UC001	**Usecase name**	Social login
Actor	Executor		
Pre-condition	Executor is a member of DKG		
Main flow of event	**No**	**Actor**	**Action**
	1	Executor	create encKey
	2	Executor	Query config and number of idle keys
	3	Executor	Share dealer phase
	4	Smart contract	Store the commitments and shares
	5	Smart contract	Auto update round status
	6	Excutor	Query the current round status and round information
	7	Executor	Construct the final shares and the pubkey share from the round information
	8	Executor	Share row phase
	9	Smart contract	Store the commitments and shares
	10	Smart contract	Auto update round status
	11	Smart contract	Push the complete round to the pool
Alternative flow of event	**No**	**Actor**	**Action**
	3b	Executor	Stop and wait the next time invoking

Table 4.6: Executor contributes in the DKG protocol

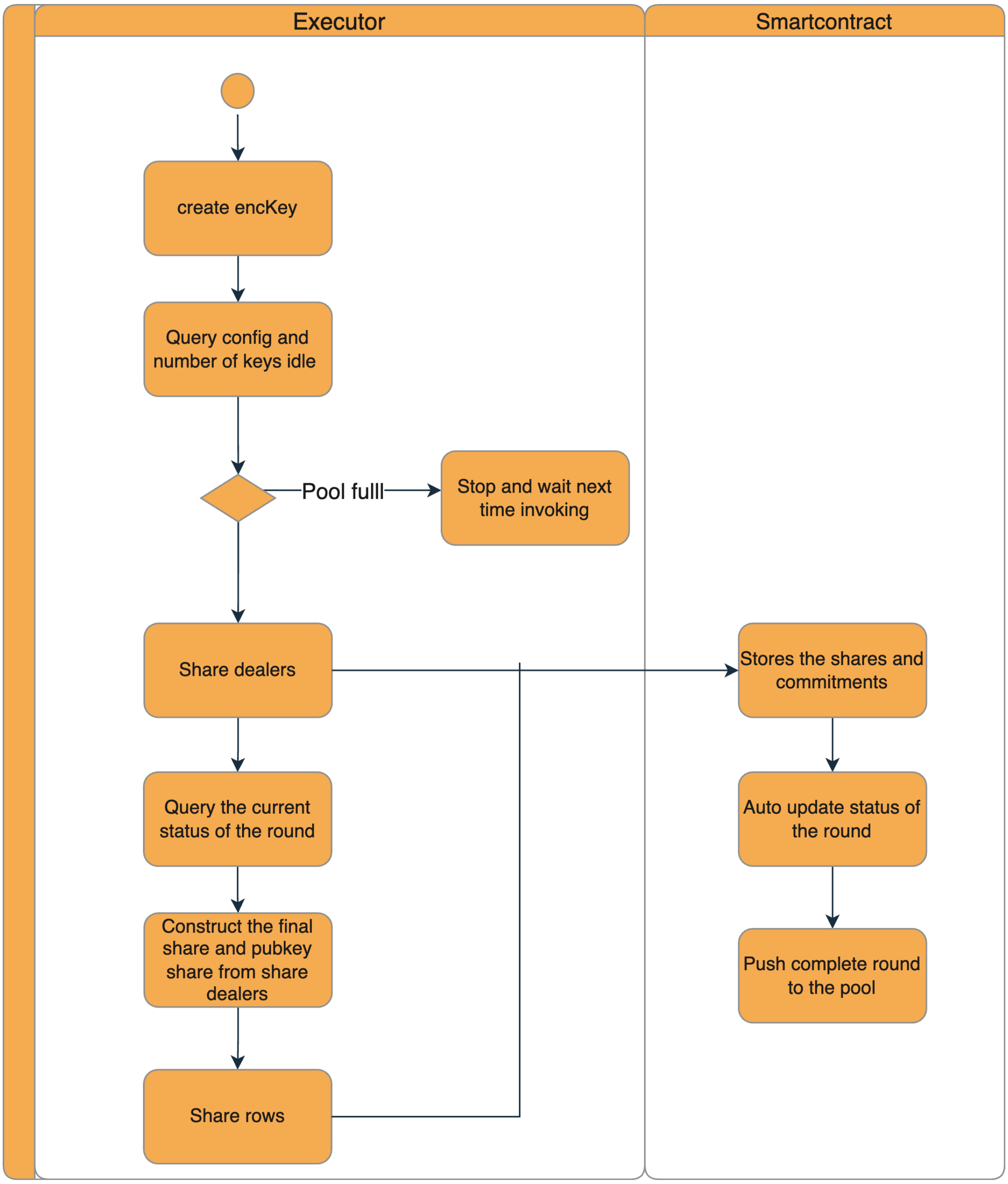

Figure 4.5: Executor contributes in the DKG protocol activity diagram

The participation of an executor in the encKey creation procedure is crucial for the successful operation of the system. The executor, as a special actor, plays an active role in contributing to the system's functioning. The process involves multiple steps and interactions between the executor and the public smart contract deployed on the blockchain.

Table 4.6 provides a comprehensive overview of the executor's involvement in the encKey creation procedure. It outlines the specific tasks and responsibilities assigned to the executor, including initiating the encKey creation round, verifying the round status, aggregating shares from other participants, generating and encrypting their own share, and finally submitting the encrypted shares to the smart contract.

Figure 4.5 complements the table by presenting an activity diagram that illustrates the sequence of actions performed by the executor during the encKey creation process. The diagram highlights the points of interaction between the executor and the public smart contract, depicting the flow of control and data between the two entities.

Usecase code	UC001	**Usecase name**	Social login
Actor	Executor		
Pre-condition	Executor is a member of DKG		
Main flow of event	**No**	**Actor**	**Action**
	1	Executor	Assign key
	2	Executor	Send idToken or accessToken to Auth0
	3	Auth0	Verify token
	4	Auth0	Send verify responses
	5	Executor	Validate the user information
	6	Executor	Send the signatures for the client for aggregate
	7	Executor	Validate signatures
	8	Executor	Send request assign key with signatures to smart contract
	9	Smart contract	Process the request
Alternative flow of event	**No**	**Actor**	**Action**
	6b	Executor	Reject request if invalid user information
	8b	Executor	Reject request if the valid signatures is not exceed the thresholds

Table 4.7: Executor assigns key

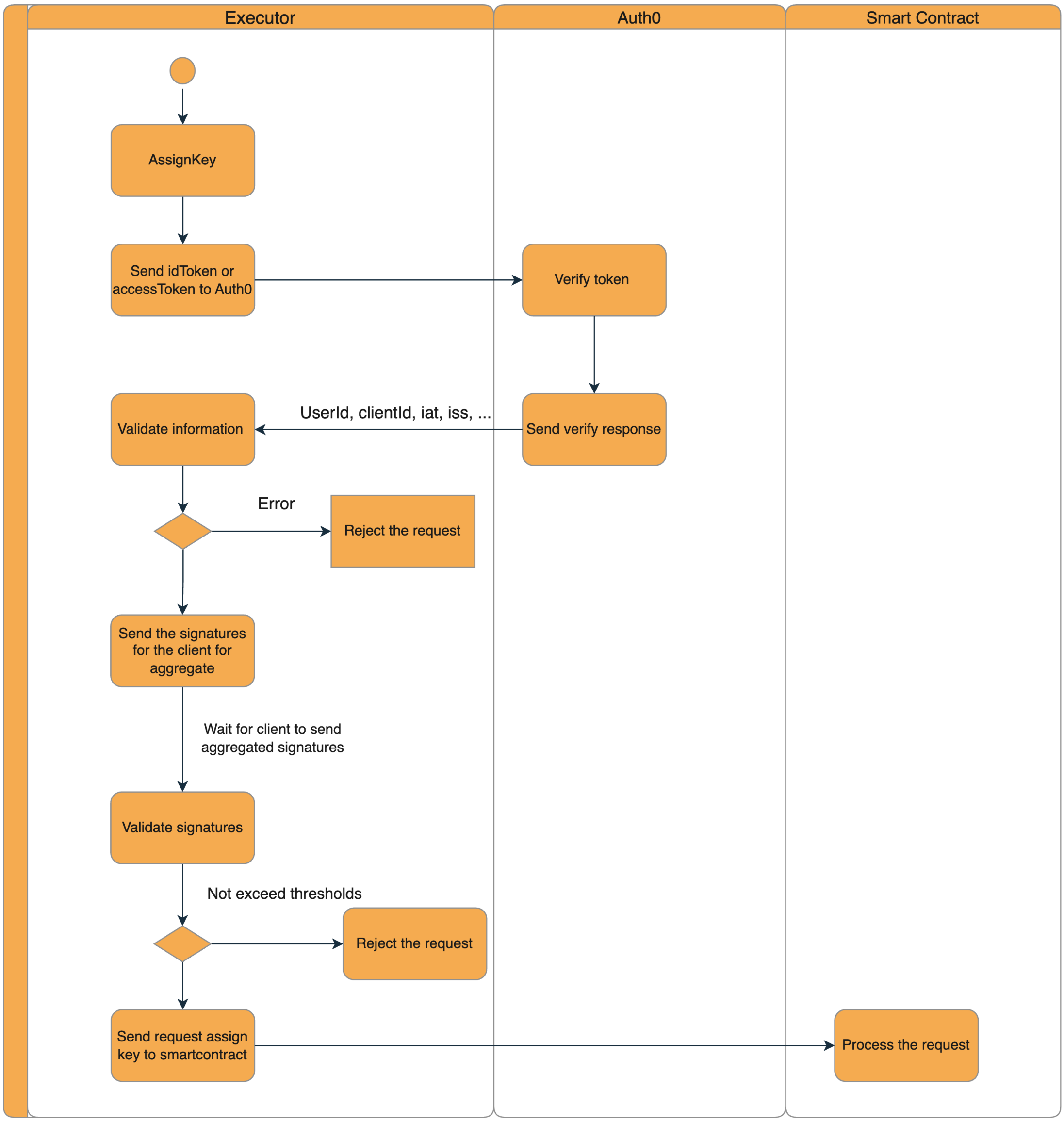

Figure 4.6: View private key and shares description activity

Figure 4.6 provides a detailed illustration of the process in which an executor assigns an encKey to a user in the social login system for Web3. This diagram showcases the interactions between three key components: Auth0, the smart contract, and the executor.

4.2 Technologies

In addition to the blockchain technology, smart contracts, and oracles described in the background section, I would like to describe other system-building tools and libraries.

4.2.1 Backend

4.2.1.1 ExpressJs

Express.js is a well-known web application framework for Node.js that facilitates the construction of scalable and robust web applications. It offers a minimalistic and adaptable approach to web development, making it simple for developers to construct server-side applications.

One of the primary benefits of Express.js is its simplicity and usability. It provides developers with a framework that is minimal and agnostic, allowing them to structure their applications according to their needs. Express.js adheres to a middleware-based architecture, enabling developers to easily incorporate middleware components to handle various application aspects, such as routing, request processing, and error handling.

Express.js [17] is widely recognized for its robust routing capabilities. It offers a concise and straightforward syntax for defining routes and managing various HTTP methods, including GET, POST, PUT, and DELETE. Express.js makes it simple to define routes for handling incoming requests and performing operations such as retrieving data from a database, processing the data, and returning the appropriate response to the client.

Within the system architecture, Express.js serves as a robust and versatile server framework that plays a vital role in handling the requests associated with the metadata of the system. Leveraging the power of Express.js, the backend of the system is equipped with the capability to receive, process, and respond to various types of requests related to the metadata. Overall, the utilization of Express.js as the server framework for handling metadata requests empowers the system to efficiently manage and process metadata-related operations. Its robust routing system, extensible middleware ecosystem, and high-performance nature make Express.js an ideal choice for building scalable and reliable backend systems that handle metadata effectively.

4.2.1.2 JSON-RPC and Jayson library

JSON-RPC[18] is a remote procedure call protocol encoded in JSON. It allows for the execution of remote procedures on a server and the exchange of structured data between the client and the server. In the system, JSON-RPC is employed to facilitate communication between different components and services, enabling the invocation of remote procedures and the transmission of data in a standardized and efficient manner. The utilization of JSON-RPC ensures interoperability and seamless integration between various system components.

The Jayson library, a JSON-RPC implementation for Node.js, is used to manage JSON-RPC communication between the client and server. Jayson is a framework for creating JSON-RPC servers and clients in Node.js that is lightweight and efficient. It simplifies the creation and management of JSON-RPC requests and responses, facilitating integration with the server and client components of the system. You can define the server-side methods that will be exposed to clients via JSON-RPC using Jayson. Jayson handles the serialization and deserialization of JSON-RPC messages. These methods can be implemented as functions or class methods. It offers a convenient API for managing the request payload, extracting the method name and parameters, and executing the appropriate server-side logic. The Jayson library integrates seamlessly with the Express.js server, allowing the server component of the system to process incoming JSON-RPC requests using the specified server-side methods. It offers an elegant and efficient solution for implementing JSON-RPC communication in Node.js, making it an excellent choice for the communication layer of the system.

The Jayson library is incorporated in the system to process JSON-RPC requests from the client to the executor component. This decision is made because the role of the executor predominantly entails handling and processing requests unrelated to database operations. Using Jayson for JSON-RPC communication allows the executor to efficiently manage requests without requiring direct database interaction. JSON-RPC requests may include information and instructions for the executor to perform particular duties or computations, such as generating shares, decrypting data, or validating signatures. The executor processes these requests, carries out the required logic, and returns the appropriate response to the client. Implementing JSON-RPC for the executor enables a clean separation of concerns, as the executor component is exclusively focused on executing the requested tasks and does not interact directly with the databases. This design decision streamlines the executor's responsibilities and improves its capacity to respond to client requests. Using a well-established library such as Jayson also facilitates the implementation of JSON-RPC communication, resulting in a robust and dependable solution. The library manages the serialization and deserialization of JSON-RPC messages, ensuring that requests and responses are formatted correctly and are compatible between the client and executor. This simplifiees the development process and helps keep the communication protocol consistent and compatible.

4.2.2 Database

MySQL[19] is an extensively utilized open-source relational database management system (RDBMS) for storing and querying structured data. In your system, MySQL is used exclusively for storing and querying metadata in the metadata backend section. MySQL offers a dependable and effective method for storing structured data in tables, with support for a variety of data types and indexing options. It provides comprehensive transactional capabilities, ensuring data consistency and integrity during concurrent operations. MySQL's robust query language, SQL (Structured Query Language), enables efficient data retrieval and manipulation through a variety of query operations, including filtering, sorting, joining, and aggregating. The metadata backend section of the system is responsible for managing and storing metadata associated with user authentication, authorization, and other pertinent data. MySQL functions as the underlying database technology for the structured storage of this metadata. In the future, the metadata may include encrypted shares or ecrypted information of multiple system modules. By leveraging MySQL, the system can take advantage of its scalability and performance enhancements, enabling it to efficiently manage large volumes of metadata. In addition, MySQL provides security mechanisms, data backup and recovery, and replication options, all of which contribute to the dependability and accessibility of your metadata storage.

4.2.3 Frontend

The frontend of the system plays a critical role as it enables users or clients to interact with the system in a seamless and user-friendly manner. It serves as the interface through which users can access the system's features, functionalities, and data.

Tkey is a GitHub-hosted open-source library designed to provide cryptographic key management capabilities. The library seeks to simplify the handling and management of cryptographic keys, making it easier for application developers to implement robust security measures. The Tkey library provides a vast array of key generation, storage, and usage-related features and functions. It supports a variety of cryptographic algorithms, such as symmetric and asymmetric encryption, digital signatures, and hash functions. This enables developers to employ various cryptographic techniques based on their particular requirements. Tkey's emphasis on security is one of its primary features. The library employs key management best practices, such as secure key storage, protection against unauthorized access, and secure key sharing protocols. Tkey adheres to industry standards and recommendations to assure the privacy, integrity, and accessibility of cryptographic keys. In addition, Tkey provides a simple and intuitive API, easing the integration of cryptographic key management into applications. The library provides developers with comprehensive documentation and implementation examples. Based on the foundation provided by the Tkey open-source library, our system extends its functionality by implementing modules that are specifically tailored to our application's needs. These custom modules enhance Tkey's compatibility with our system's architecture by enhancing its primary features.

4.2.4 Virtualization

Docker is an open-source platform that has revolutionized the way applications are developed, shipped, and deployed. It provides developers with a powerful set of tools to automate the packaging and deployment of applications inside lightweight, portable containers. By leveraging containerization technology, Docker enables the creation of isolated environments that encapsulate an application and its dependencies. This ensures that the application runs consistently and predictably across different systems, regardless of the underlying infrastructure.

Containers are self-contained units that contain everything needed to run an application, including the code, runtime, system tools, and libraries. They offer advantages such as isolation, scalability, and reproducibility. With Docker, developers can package their applications along with all the necessary dependencies into a container image, which can then be deployed on any system that has Docker installed. This eliminates the need for complex setup processes and reduces compatibility issues between different environments.

Docker Compose is a complementary tool to Docker that simplifies the management of multi-container applications. It allows developers to define and configure multiple containers, their networks, and volumes using a simple YAML file. This makes it easier to define complex, interconnected services and orchestrate their deployment. With Docker Compose, developers can quickly spin up an entire application stack with a single command, making the development and testing process more efficient.

In the context of this thesis, Docker plays a crucial role in encapsulating the application and its dependencies, ensuring portability and consistency. By using Docker, the application can be packaged into a container image, making it highly portable and enabling easy deployment across different environments. Docker-compose is utilized during the development phase to define and manage the various services and dependencies required by the application. This allows for rapid system configuration and simplifies the process of setting up a development environment.

4.2.5 Rust, wasm and cosmwasm

Rust is a systems programming language known for its emphasis on performance, reliability, and safety. It offers a high level of control over system resources while providing memory safety guarantees through its ownership and borrowing system. Rust's combination of low-level control and high-level abstractions makes it well-suited for building efficient and secure software, including blockchain applications.

WebAssembly (Wasm) is a binary instruction format that allows for the execution of code on a variety of platforms, including web browsers. It provides a portable and efficient runtime environment for running code at near-native speeds. Rust is one of the languages that can compile to WebAssembly, allowing developers to leverage Rust's performance and safety features in web-based applications.

CosmWasm is a specific implementation of WebAssembly designed for smart contract development within the Cosmos ecosystem. It extends the capabilities of WebAssembly to support the unique requirements of blockchain-based smart contracts. With CosmWasm, developers can write smart contracts in Rust and compile them to WebAssembly, leveraging the language's safety guarantees and performance optimizations. CosmWasm provides a secure execution environment for these smart contracts, ensuring the integrity and correctness of their execution on the blockchain.

The combination of Rust, WebAssembly, and CosmWasm offers several benefits for blockchain development. Rust's memory safety features help mitigate vulnerabilities and potential exploits in smart contracts, reducing the risk of security breaches. WebAssembly enables efficient execution of the compiled Rust code across different platforms, allowing for interoperability and portability. CosmWasm, as an implementation of WebAssembly tailored for blockchain smart contracts, provides a secure and optimized environment for executing Rust-based smart contracts within the Cosmos ecosystem.

4.3 Diagram design

4.3.1 Sequence diagram

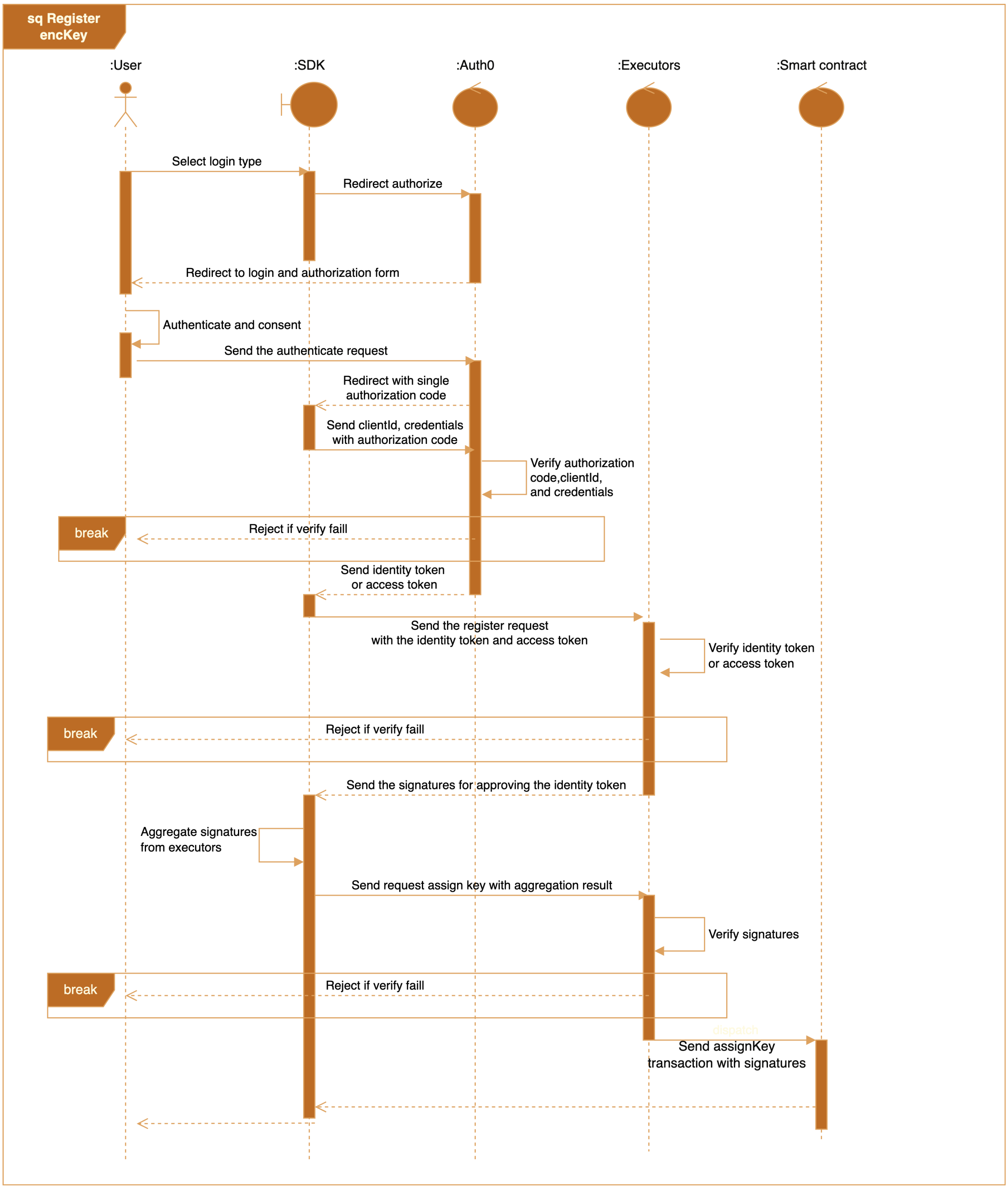

Figure 4.7: Register sequence diagram

The diagram 4.7 depicts the registration procedure for a new encKey. The user must select the social registration type and successfully complete the validation procedure.

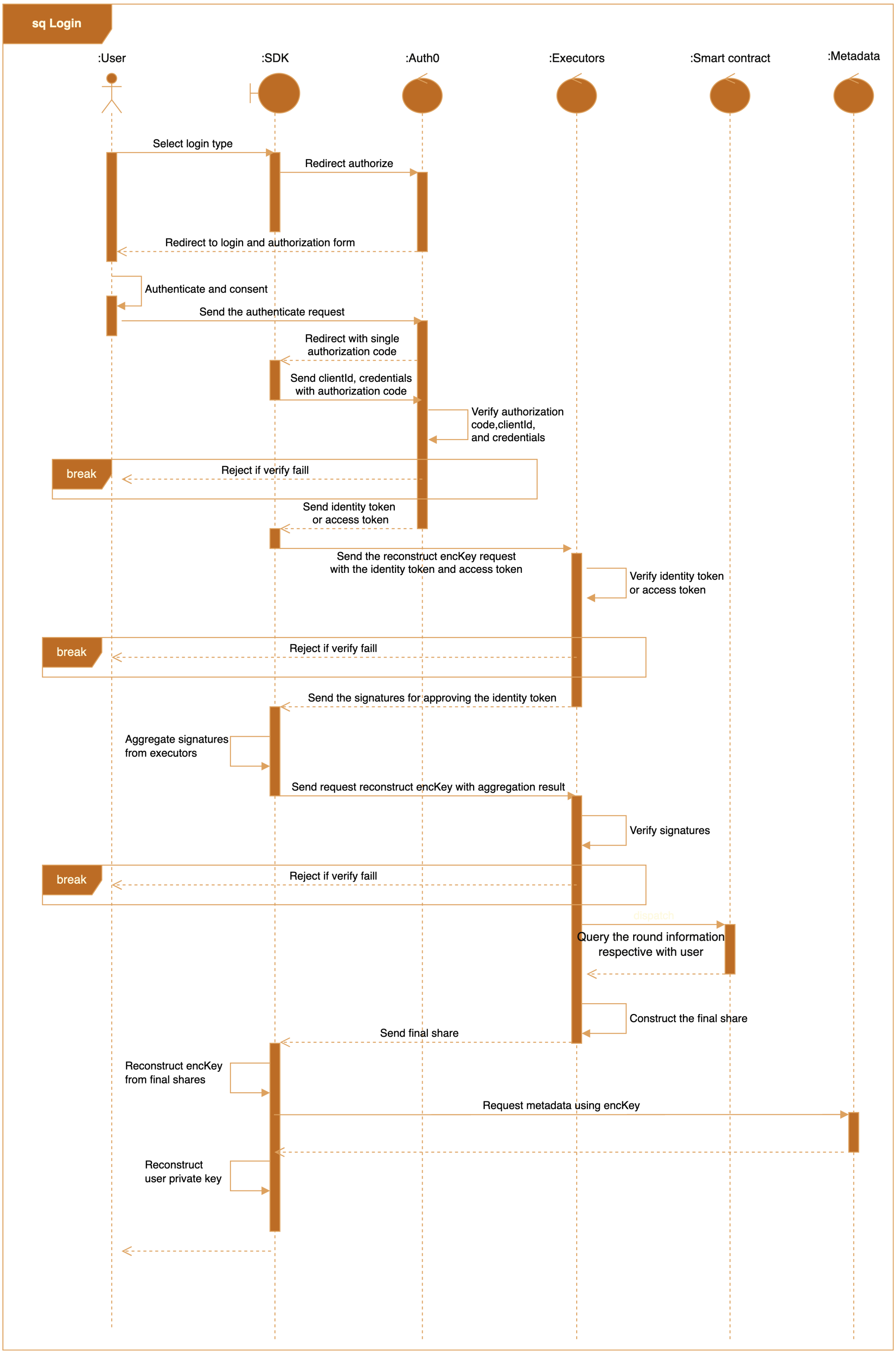

Figure 4.8: Login sequence diagram

The diagram 4.8 depicts the social authentication system's sign-in procedure. This procedure describes how the user's data pass validation and how the user reconstructs their private key.

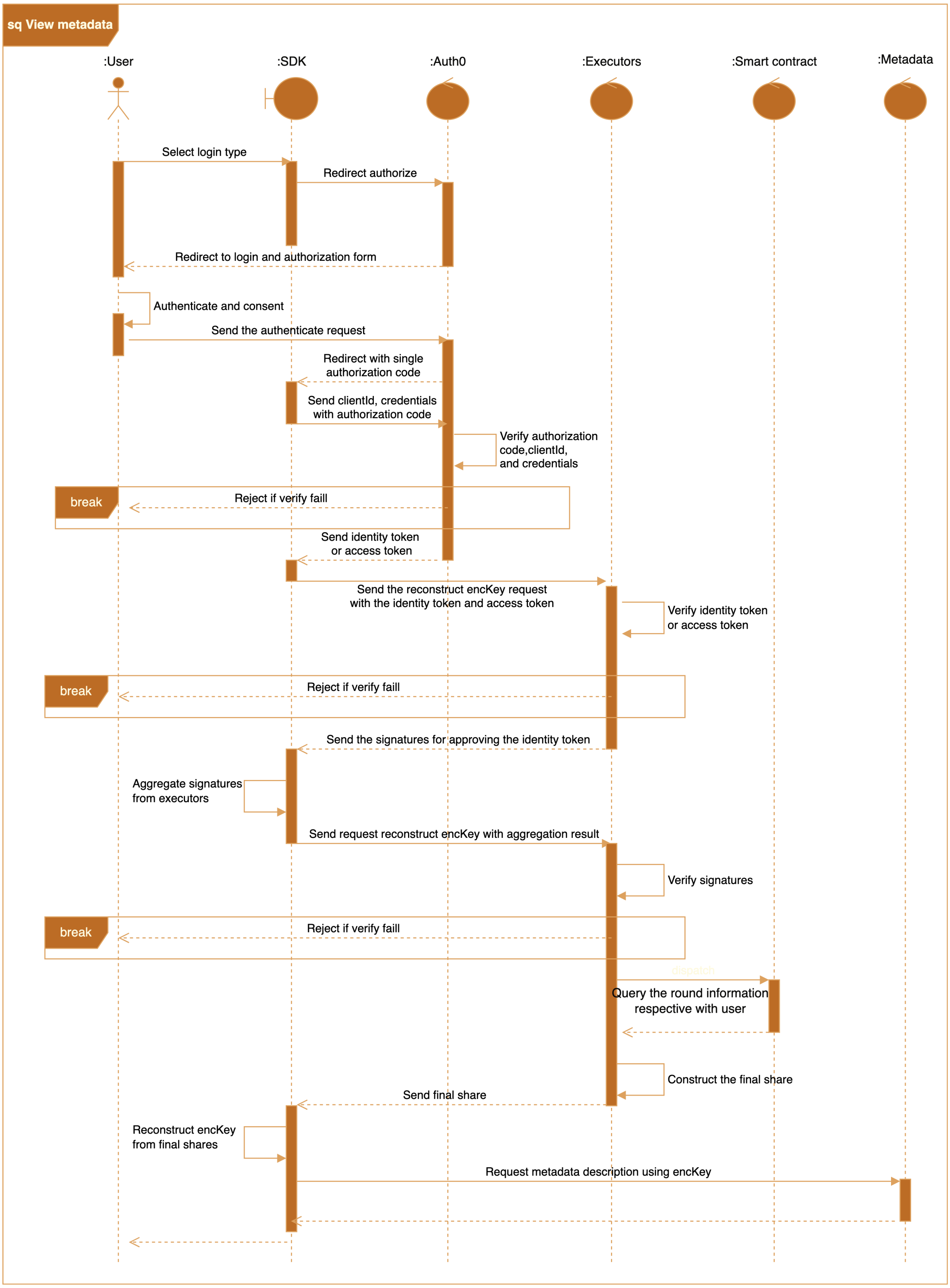

Figure 4.9: View share description sequence diagram

After logging into the system, users desire to view their share descriptions and private keys. This procedure is depicted graphically in the diagram 4.9.

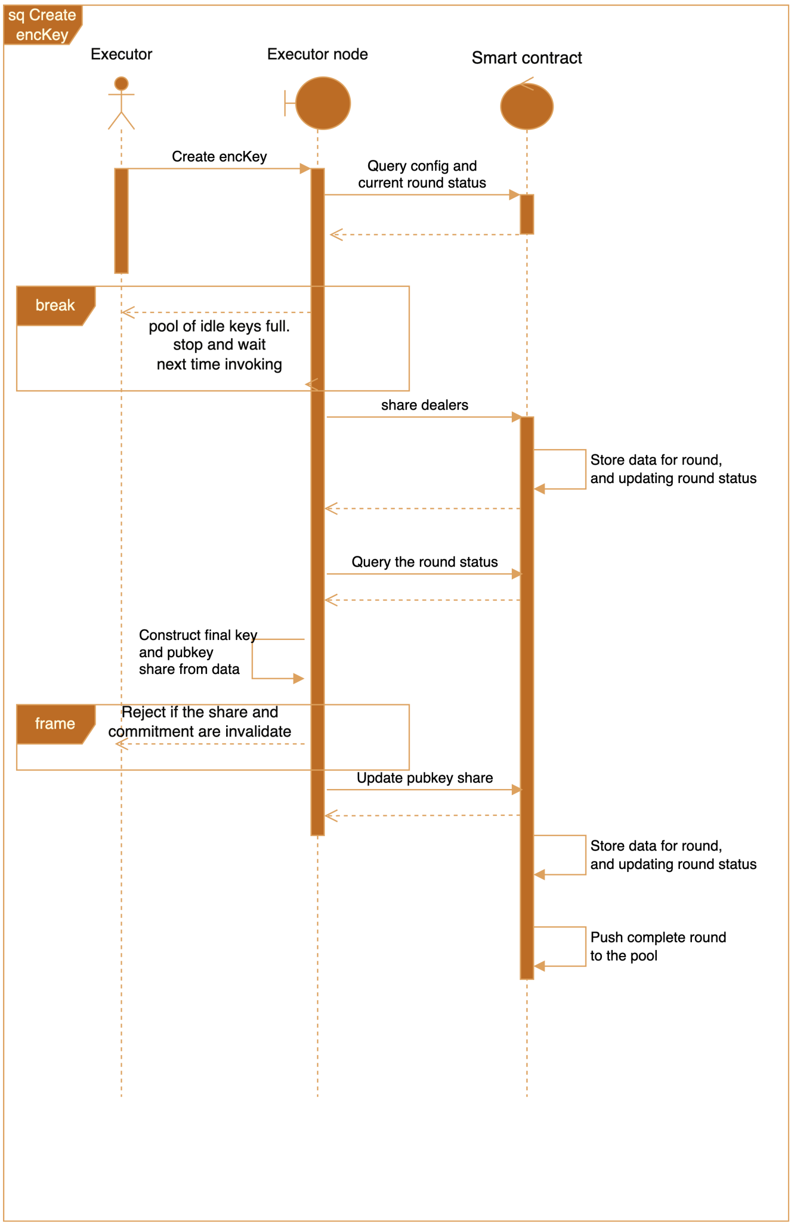

Figure 4.10: Create encKey sequence diagram

Creating a new encKey in the system is a fundamental and crucial operation that has a substantial impact on the system's overall efficacy and security. This crucial procedure is initiated when the executor, acting as a specialized actor in the system, performs the required action. To optimize the role of the executor, I meticulously designed the executor node as a cron job executor, assuring seamless management and efficient task execution. By implementing the executor node as a cron job executor, I've ensured the creation of new encKeys is optimized for efficiency and seamless administration. The cron job scheduler enables the executor to perform repetitive duties automatically and at specified intervals. This ensures that the generation of new encKeys is consistent, efficient, and always ready to satisfy the system's demands.

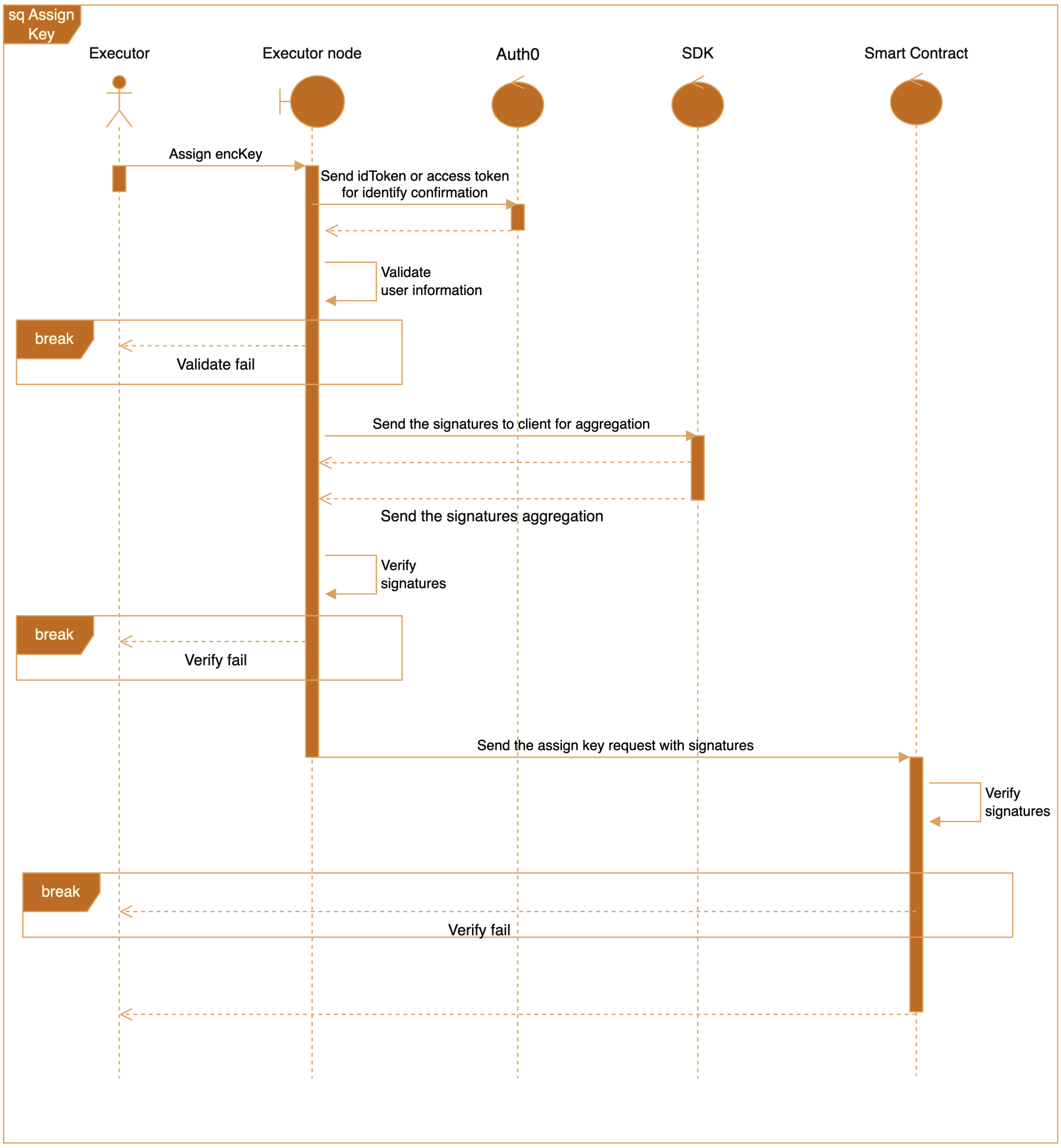

Figure 4.11: Assign encKey for user sequence diagram

The complexity and security of the executor's procedure for allocating a new encKey to a user exemplifies the robustness of the system. When a user requests their encKey, the executor receives their data, which contains crucial validation information. Through a series of exhaustive validation tests, the executor ensures that the user's data is valid, accurate, and in accordance with the system's specifications. This validation phase is a crucial line of defense against possible data inaccuracies and unauthorized access attempts. The executor verifies the signatures of other executors who participate in the assignment procedure to increase the process's security. These signature verifications are necessary to affirm the credibility and authenticity of each executor involved in the assignment process. The system ensures that only authorized and trustworthy parties contribute to the designation of the new encKey by establishing trust among all participating executors.

4.3.2 Package diagram

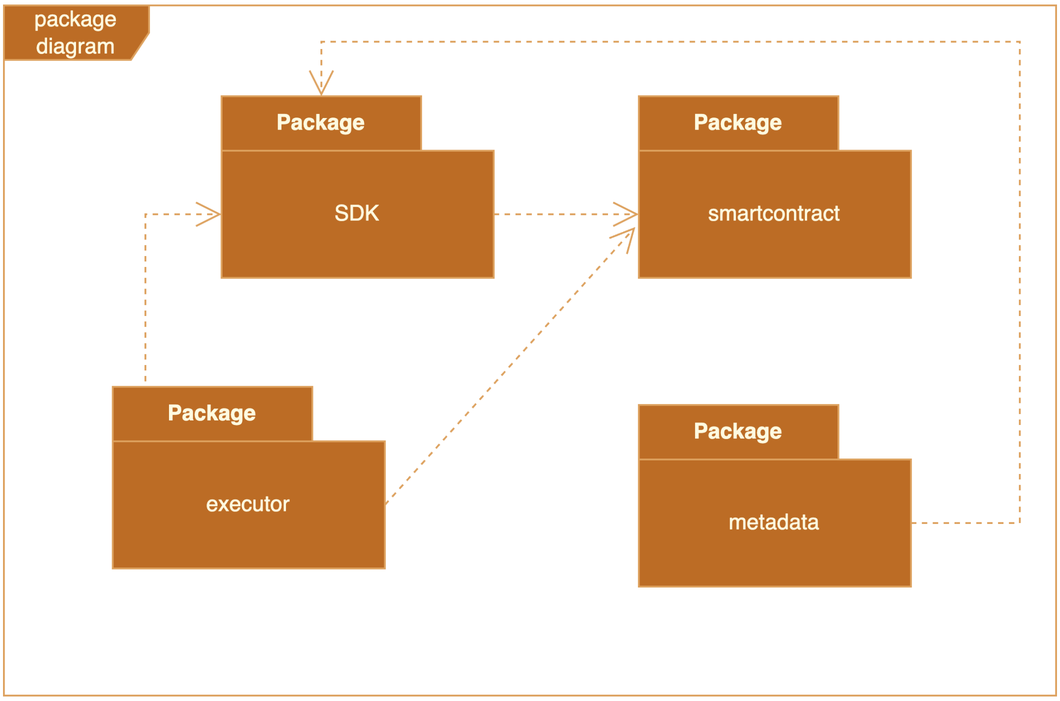

Figure 4.12: Package diagram

The system consists of multiple packages that collaborate to provide a secure and seamless user experience. The SDK package functions as the user interface, allowing users to interact with the system, view their private keys and shares, and execute various operations. The executor package functions as an intermediary between clients and the system, processing user requests and executing actions on their behalf. This package is in charge of generating new encKeys, validating user information, and interacting with the blockchain. In the meantime, the metadata package is indispensable for securely storing and querying metadata associated with user identities and encKeys. It administers user data in a database, ensuring the integrity and security of data. The interdependencies between these products are crucial to the overall functionality of the system. The SDK package relies on config data in the smart contract, which provides crucial user interaction information. The actions of the executor package are dependent on both the data contained in the smart contract and the SDK requests. This enables the executor to perform required actions while adhering to the smart contract's rules and conditions. The metadata package, on the other hand, predominantly stores and retrieves query responses from the SDK. It serves as the data storage layer for user-related information and plays a crucial role in providing efficient and secure access to user data. By meticulously orchestrating the interactions between these packages, the system is able to provide a user-friendly and robust experience, combining the advantages of blockchain technology with the convenience of social login and user-controlled encryption keys.

4.3.3 Entity diagram

An Entity-Relationship (ER) diagram depicts the relationships between database entities in a centralized system. The ER diagram illustrates how data is organized and related, allowing for a comprehensive comprehension of the data model. However, in my system the database only uses key-value relations in the metadata module. Therefore, in the context of this thesis, I would like to expand upon the normal ER-diagram's description of the entity storing in the smart contract and the relationship between them.

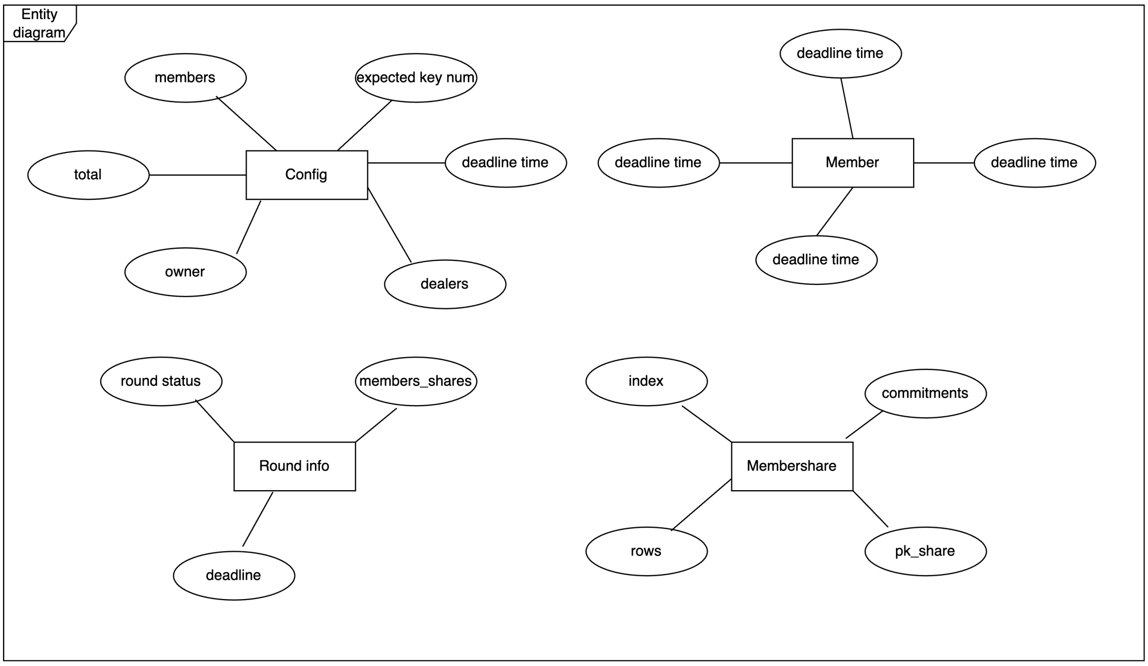

Figure 4.13: Entity diagram

All the strong entities that are stored directly as states are underlined with unique keys in the diagram. Since states are stored using key-value pairings, similar to a document-oriented database, it is unnecessary to have relationships between the entities.

The entity particulars are detailed in the tables that follow:

Attribute	Type	Required	Description
index	number	yes	The index of the member in the system plays vital role in reconstructing the encKey
pub_key	Binary	yes	The pubkey of the executor participating in the system
address	Addr	yes	The address of the executor participating in the system
end_point	string	yes	end_point is the url to executor node, which for the request in the SDK

Table 4.8: Member entity detail

Attribute	Type	Required	Description
members	Vec<Member>	yes	The member or executors operate the system
total	number	yes	Total member participate in the system
owner	Addr	yes	The address of the owner of the system
expected_key_num	number	yes	The number of expected keys are in idle pool
deadline_time	number	yes	The maxium duration of creating a key
dealers	number	yes	Thresholds for the 2 phase "Wait for dealers" and "Wait for rows"

Table 4.9: Config entity detail

Attribute	Type	Required	Description
index	number	yes	The index of executor in the system
commitments	Vec<Binary>	no	The commitments of shares of executor for the rows
rows	Vec<Binary>	no	The encrypted shares for other executors
pk_share	Binary	no	The pubkey of the final share of executor

Table 4.10: Membershare entity detail

Attribute	Type	Required	Description
round_status	number	yes	The status of the round, there are 4 status include in: WaitForDealers, WaitForRows, WaitForAssign, and Assigned
member_shares	Vec<Membershare>	no	The shares of executor for round
deadline	Vec<Binary>	no	The deadline of this round

Table 4.11: RoundInfo entity detail

4.4 Message design

4.4.1 Smart contract message

CosmWasm [20] is a smart contract platform that has been developed utilizing the Cosmos SDK[21], a blockchain framework specifically designed for the construction of DApps. CosmWasm offers a robust and optimized framework for the execution of intelligent contracts across diverse blockchain networks, ensuring both security and efficiency. The CosmWasm smart contract model is founded upon the actor model, a widely adopted paradigm utilized in the design of concurrent and distributed systems. The actor model is a computational paradigm in which each smart contract is conceptualized as an actor. An actor is an autonomous entity capable of receiving messages, performing computations on them, and transmitting messages to other actors. In the CosmWasm smart contract model, the actors are characterized by their isolation from one another, with communication occurring exclusively through the exchange of messages.

4.4.1.1 Instantiate message

The concept of instantiation messages pertains to the process of instantiating and deploying a smart contract on the blockchain.

Field	Type	Required	Description
members	Vec<Member>	yes	The information of executors participate in the system
dealers	u8	yes	Thresholds of the system for passing each phase
owner	string	yes	Owner of the smart contract
expected_key_num	uint	yes	The expected number of key in idle pool
deadline_time	uint	yes	The maxium duration of 1 round

Table 4.12: Instantiate message detail

4.4.1.2 Execute message

The execute message serves as the primary interface for processing incoming messages and executing the contract's logic in accordance with the provided input. Every execute message contained within the executor message enumeration possesses a distinct structure and data payload, thereby empowering the smart contract to effectively react to a wide array of interactions originating from users or other entities within the blockchain ecosystem.

The following tables provide a more comprehensive description of each execute message:

Field	Type	Required	Description
rows	Vec<Binary>	yes	The shares given to each participant in the DKG protocol in order to construct their final share for encKey.
commitments	Vec<Binary>	yes	The commitments are to confirm that the rows originate from the precise source

Table 4.13: ShareDealerMsg detail

Field	Type	Required	Description
pk_share	Binary	yes	pk_share is the pubkey of the final share that the executor reconstructs using rows and commitments from the share dealer.

Table 4.14: ShareRowMsg detail

Field	Type	Required	Description
members	MemberMsg	no	The member config
owner	string	no	The new owner
dealers	u8	no	The new thresholds
expected_key_num	uint	no	The new expected number of key in idle pool
deadline_time	uint	no	The new maxium duration of key creation

Table 4.15: UpdateConfigMsg detail

Field	Type	Required	Description
sigs	Vec<Binary>	yes	The signatures of the executor which approve for assigning key for user
pub_key	Vec<Binary>	yes	The pubkey of the executors that create the signature
verifier	string	yes	The name of app which is whitelisted in the system
verifier_id	string	yes	email or user_id

Table 4.16: AssignKeyMsg detail

Field	Type	Required	Description
verifier	string	yes	The name of app which is whitelisted in the system
client_id	string	yes	The client_id of the app

Table 4.17: UpdateVerifierMsg detail

4.4.1.3 Query message

QueryMsg is a specific variety of execute message within the executor message enum within the context of the CosmWasm smart contract and actor model. It is designed to manage query requests and enables the smart contract to provide read-only, non-modifiable access to its current state. When a query request is received, the smart contract processes the QueryMsg and executes the required operations to retrieve the requested information from its internal state. As a response to the inquiry, the smart contract can then construct and return the requested data.

The query messages and responses schema are desbribed in detail by the following tables:

Field	Type	Required	Description
round_id	uint	yes	The id of the round
round_status	usize	yes	The status of the round
members_shares	Vec<MemberShare>	yes	The details about the membershares of the round
deadline	uint	yes	The deadline of the round

Table 4.18: RoundInfoResponse detail

Field	Type	Required	Description
members	Vec<Member>	yes	The information of the executor participating in the system
total	uint	yes	The number of the executor
dealer	uint	yes	Thresholds of the system
owner	string	yes	Owner of the system
expected_key_num	uint	yes	The number of expected keys in idle pool

Table 4.19: ConfigResponse detail

Field	Type	Required	Description
verifier	string	yes	The app that need to query user email
start_after	string	yes	The exclude bound start of the query
limit	uint	no	Number of verifierId. Default is 10

Table 4.20: ListVerifierIdMsg detail

Field	Type	Required	Description
verifier	string	yes	The app that need to query user email
list_verifier_id	Vec<string>	yes	List of the email or user_id of the app

Table 4.21: ListVerifierIdResponse detail

Field	Type	Required	Description
msg	Binary	yes	The mesage that was signed by executor
sigs	Vec<Binary>	yes	Signatures for the message
pub_keys	Vec<Binary>	yes	The public key of executors signed the message

Table 4.22: VerifyMemberMsg detail

4.4.2 JRPC message

The executor plays a crucial role in the creation and distribution of encryption keys to users. The executor exposes four essential JSON-RPC (JRPC) methods to facilitate this functionality:

AssignKeyCommitmentRequest - The executor uses this JRPC method to request the user's commitment to a new encryption key. The user will commit to the key without disclosing its actual value. This phase verifies that the user intends to generate a valid encryption key.

AssignKeyRequest - The executor uses this JRPC method to request the actual encryption key from the user after obtaining the user's commitment. The user will provide the encrypted key in accordance with the commitment, ensuring that the key remains secret until later phases of the process.

CommitmentRequest - The executor uses this JRPC method to request the commitments made by other executors in the system. To proceed with the encryption key generation process and accomplish the desired level of security and consensus, other executors' commitments are required.

ShareRequest - Lastly, the executor uses the ShareRequest JRPC method to request shares from other executors in order to reconstruct the encryption key. As part of the distributed key generation protocol, the executor must obtain portions from multiple other executors.

The subsequent tables illustrate in depth the specifics of message and response:

Field	Type	Required	Description
jsonrpc	string	yes	The version of JRPC. Must be "2.0"
method	string	yes	The method will call in the json-rpc server
id	number	yes	An identifier established by the Client
params	Object	yes	A Structured value that holds the parameter values to be used during the invocation of the method

Table 4.23: General JPRC request detail

Field	Type	Required	Description
jsonrpc	string	yes	The version of JRPC. Must be "2.0"
result	string	no	This member is REQUIRED on success. This member MUST NOT exist if there was an error invoking the method. The value of this member is determined by the method invoked on the Server.
id	number	yes	An identifier established by the Client
error	Object	no	This member is REQUIRED on error. This member MUST NOT exist if there was no error triggered during invocation. The value for this member MUST be an Object

Table 4.24: General JPRC response detail

The subsequent tables will describe in detail the request parameters and the response value:

Field	Type	Required	Description
tokencommitment	string	yes	The hash of the idToken or access token receive from Autho0
temppubX	string	yes	The pubkey in X axis of temp key established in the reconstruct process
tempubY	string	yes	The pubkey in Y axis of temp key established in the reconstruct process

Table 4.25: CommitmentRequest parameters detail

Field	Type	Required	Description
signature	string	yes	The signature of executor sign on data
data	string	yes	The data is signed by executor
nodepubX	string	yes	The pubkey in X axis of executor node using for verify
nodepubY	string	yes	The pubkey in Y axis of executor node using for verify

Table 4.26: CommitmentRequest response value detail

Field	Type	Required	Description
verifier	string	yes	The name of the app that registered in the system
verifier_id	string	yes	The email or user_id of user
idToken	string	yes	The idtoken of the Auth0
nodesignatures	Object	yes	The result of the commitment request

Table 4.27: ShareResponseRequest parameters detail

Field	Type	Required	Description
Publickey	string	yes	The publickey of the first commitment
Share	string	yes	The share encrypted with the client's public temporary key. This portion is used to reconstruct encKey.
Metadata	Object	yes	The description of the share, which is necessary for decrypting the share

Table 4.28: ShareRespone result value detail

Field	Type	Required	Description
tokencommitment	string	yes	The hash of the idToken or access token receive from Auth0
verifier	string	yes	The app that using the system
verifier_id	string	yes	Email or useri_id respectively in the app

Table 4.29: AssignKeyCommitmentRequest parameters detail

Field	Type	Required	Description
data	Object	yes	The publickey of the first commitment
nodepubx	string	yes	The publickey in X-axis of executor
nodepuby	string	yes	The publickey in Y-axis of executor
signature	string	yes	The signature of the data in stringtify
verifierIdSignature	string	yes	The signature was signed by executor about the userid or email

Table 4.30: AssignKeyCommitmentResponse value detail

Field	Type	Required	Description
idToken	string	yes	The publickey of the first commitment
verifier	string	yes	The publickey in X-axis of executor
verifier_id	string	yes	The publickey in Y-axis of executor
nodesignatures	Object	yes	The response of the assignKeyCommitmentResponse in stringtify

Table 4.31: AssignKeyRequest parameter detail

Field	Type	Required	Description
status	bool	yes	The response of the assignKeyRequest. If the status fail, the system will retry until 5 times.

Table 4.32: AssignKeyResponse value detail

4.4.3 HTTPS request

The HTTPS request is an integral element of the system's communication between the client and the metadata server. It allows the client to perform CRUD operations on the server's database-stored metadata. The HTTPS protocol is used to ensure secure and encrypted communication between the client and the server, thereby safeguarding data and guaranteeing the privacy and integrity of transactions. When a client needs to create new metadata, it transmits an HTTPS POST request containing the required information to the server. The server processes the request, verifies the data, and adds the newly generated metadata to its database. The client sends an HTTPS GET request to the server for reading data, specifying the unique identifier of the metadata or any other criteria for retrieving specific entries. The server retrieves the requested data from the database and returns it in response to the client.

The tables below illustrate in detail about the messages and responses of the HTTPS request:

Field	Type	Required	Description
pub_key_X	string	yes	The pubkey in X-axis of encKey or the share of client
pub_key_Y	string	yes	The pubkey in Y-axis of encKey or the share of client
set_data	Object	no	The data will be store in the metadata server. If not exist, the request will be the query request
signature	string	yes	the signature of the data that will be store in the server
namespace	string	no	The namespace will store in the database. Default is "tkey"

Table 4.33: The https request detail

Field	Type	Required	Description
status	bool	yes	Status of the request to the server.
metadata	Object	no	If the request is made using the GET method, the server's response will be the metadata.

Table 4.34: The https response detail

5.1 Testing

This section will analyze the performance of the four kind of JRPC request that are essential for the login and registration use cases of the system. The requests are crucial for facilitating communication between the SDK and executors, and their performance significantly affects the overall efficiency and user experience of the system.

The four JRPC requests being analyzed are AssignKeyCommitmentRequest, AssignKeyRequest, CommitmentRequest, and ShareRequest. To assess the performance of JRPC requests, we will conduct thorough testing and gather data on response times, request rates, and throughput. The objective is to assess the system's performance under different loads and detect any bottlenecks or areas that require enhancement. Furthermore, we will assess the influence of various factors, including network latency and system resources, on the performance of JRPC requests.

The performance analysis will offer valuable insights into the system's scalability and its capacity to handle a high volume of users and concurrent requests. Optimizing the performance of critical JRPC requests can enhance system responsiveness, reduce latency, and improve the user experience during login and registration.

To conduct tests, I utilize the autocannon[22] package, which can be executed directly in Node.js with the following hyperparameters:

- Hyperparameters for autocannon:
  - connections: 10, 100, 1000
  - durations: 10s
  - number of worker: 1
- Hardware specification:
  - Processor: Apple M1 Pro, 6 performance cores(600 - 3220 MHz) and 2 power efficiency core(600 - 2064 MHz)
  - Memory: 16GB

5.1.1 CommitmentRequest

Stat	**2.5%**	**50%**	**97.5%**	**99%**	**Avg**	**Stdev**	**Max**
Latency	0 ms	0 ms	5 ms	8 ms	0.93 ms	2.08 ms	71 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	4073	4073	6631	8359	6850.5	1134.82	4073
Bytes/Sec	5.66 MB	5.66 MB	9.22 MB	11.6 MB	9.52 MB	1.58 MB	5.66 MB

Table 5.1: 10 connections performance

Stat	**2.5%**	**50%**	**97.5%**	**99%**	**Avg**	**Stdev**	**Max**
Latency	4 ms	9 ms	29 ms	36 ms	10.66 ms	8.48 ms	406 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	5391	5391	9247	10647	8964.4	1496.42	5391
Bytes/Sec	7.5 MB	7.5 MB	12.9 MB	14.8 MB	12.5 MB	2.08 MB	7.49 MB

Table 5.2: 100 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	26 ms	57 ms	134 ms	284 ms	97.63 ms	461.09 ms	9882 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	3837	3837	9607	10495	8381.9	2219.44	3836
Bytes/Sec	5.33 MB	5.33 MB	13.4 MB	14.6 MB	11.6 MB	3.09 MB	5.33 MB

Table 5.3: 1000 connections performance

This performance analysis examined the system's behavior across three connection scenarios: 10, 100, and 1000 connections. The evaluation metrics included latency, throughput (measured in requests per second - Req/Sec), and the frequency of timeout errors.

The system demonstrated excellent performance in terms of low latency and high throughput for both 10 and 100 connections. The average latency for 10 connections was significantly low at 0.93 ms, while for 100 connections, it remained reasonably low at 10.66 ms. The throughput for these scenarios was notable, with values of approximately 6850.5 and 8964.4 requests per second, respectively.

However, the system encountered significant challenges when the number of connections reached 1000. The average latency increased to 97.63 ms, indicating a significant delay in request processing. Furthermore, the throughput decreased to 8381.9 requests per second, indicating a diminished ability to handle the augmented workload.

The evaluation revealed a significant concern regarding the occurrence of 168 timeout errors out of 1000 connections. These errors indicate that the system experienced difficulties in handling the increased workload, resulting in delays or failures in processing requests within a reasonable timeframe.

5.1.2 ShareRequest

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	325 ms	465 ms	1130 ms	1289 ms	497.26 ms	177.23 ms	1307 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	10	10	19	28	19.5	5.38	10
Bytes/Sec	13 kB	13 kB	24.8 kB	36.5 kB	25.4 kB	7.01 kB	13 kB

Table 5.4: 10 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	1751 ms	3888 ms	5850 ms	5937 ms	3843.44 ms	990.28 ms	6059 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	0	0	19	32	20.6	9.67	12
Bytes/Sec	0 B	0 B	24.8 kB	41.8 kB	26.9 kB	12.6 kB	15.6 kB

Table 5.5: 100 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	1458 ms	5752 ms	9791 ms	9924 ms	5604.38 ms	2517.41 ms	9982 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	0	0	25	35	24.3	8.88	22
Bytes/Sec	0 B	0 B	32.6 kB	45.7 kB	31.7 kB	11.6 kB	28.7 kB

Table 5.6: 1000 connections performance

The performance statistics derived from the experiments, which involved different numbers of connections, offer valuable insights into the behavior of the system. The experiments involving 10 and 100 connections demonstrated consistent performance without encountering any errors. In the scenario involving 1000 connections, the system encountered a significant difficulty, resulting in 700 timeout errors.
Upon analyzing the latency metric, it becomes apparent that the response times exhibited an increase in correlation with the escalation of the number of connections. In the experiment involving 1000 connections,

the latency percentiles of 97.5The system consistently maintained an average throughput of approximately 24 requests per second throughout all three experiments. In the scenario involving 1000 connections, the 97.5th percentile of throughput reached a stable value of 35 requests per second. The system's capacity to handle requests was generally consistent, with the exception of a few instances where slower processing times were observed under higher loads.

5.1.3 AssignKeyCommitmentRequest

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	285 ms	305 ms	478 ms	522 ms	323.71 ms	52.36 ms	681 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	208	208	307	339	304.7	39.09	208
Bytes/Sec	301 kB	301 kB	444 kB	491 kB	441 kB	56.6 kB	301 kB

Table 5.7: 10 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	667 ms	2259 ms	4025 ms	4232 ms	2240.57 ms	676.53 ms	4521 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	237	237	432	467	399.4	80.26	237
Bytes/Sec	343 kB	343 kB	625 kB	676 kB	578 kB	116 kB	343 kB

Table 5.8: 10 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	667 ms	2259 ms	4025 ms	4232 ms	2240.57 ms	676.53 ms	4521 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	237	237	432	467	399.4	80.26	237
Bytes/Sec	343 kB	343 kB	625 kB	676 kB	578 kB	116 kB	343 kB

Table 5.9: 1000 connections performance

The throughput of the AssignKeyCommitment endpoint exhibits a positive correlation with the number of connections, as evidenced by the average request rates of 304.7, 399.4, and 578 requests per second for 10, 100, and 1000 connections, respectively. However, as the number of connections increases, the latency also increases. The average latencies for different connection counts are 323.71 ms, 2240.57 ms, and 2240.57 ms, respectively. This suggests that there may be performance bottlenecks at higher loads, which might need to be addressed through optimization.

5.1.4 AssignKey

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	447 ms	465 ms	2193 ms	2225 ms	547.96 ms	343.23 ms	2230 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	4	4	20	25	17.7	6.28	4
Bytes/Sec	3.44 kB	3.44 kB	17.2 kB	21.5 kB	15.2 kB	5.4 kB	3.44 kB

Table 5.10: 10 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	453 ms	500 ms	784 ms	834 ms	539.74 ms	96.42 ms	1154 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	96	96	185	219	180.8	35.31	96
Bytes/Sec	82.6 kB	82.6 kB	159 kB	188 kB	155 kB	30.4 kB	82.6 kB

Table 5.11: 100 connections performance

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	837 ms	1561 ms	6822 ms	8665 ms	1842.51 ms	1257.83 ms	9971 ms
Stat	**1%**	**2.5%**	**50%**	**97.5%**	**Avg**	**Stdev**	**Min**
Req/Sec	68	68	277	387	265.9	91.27	68
Bytes/Sec	58.5 kB	58.5 kB	238 kB	333 kB	229 kB	78.5 kB	58.5 kB

Table 5.12: 1000 connections performance

The average throughput in the AssignKeyRequest tests exhibited an upward trend as the number of connections increased. Specifically, the average throughput reached 17.7, 180.8, and 265.9 requests per second for tests with 10, 100, and 1000 connections, respectively. However, higher connections were found to be associated with increased latency. Specifically, average latencies of 547.96 ms, 539.74 ms, and 1842.51 ms were observed. The results emphasize the need for optimization strategies to balance throughput and latency, ensuring efficient server performance and user experience.

5.2 Evaluation

Web3Auth is a decentralized oracle network that exhibits similarities with the thesis project. The system also incorporates Shamir's Secret Sharing and DKG (Distributed Key Generation) protocol. Web3Auth provides various key features, as follows:

- **Seamless onboarding**-Web3Auth uses social login to allow users to sign up for dapps with just a few clicks. This makes it easy for users to get started with dapps, and it also helps to improve the user experience.
- **Multi-party computation (MPC)**-Web3Auth uses MPC to provide a secure and private way for users to share their keys. This allows users to collaborate on dapps without having to reveal their private keys. MPC is a cryptographic protocol that allows multiple parties to jointly compute a function without revealing their individual inputs. This makes it a very secure way for users to share their keys, as only the final result of the computation is revealed.

Web3Auth has the following disadvantages:

- The implementation of nodes for constructing keys of Web3Authen is complex due to the presence of multiple layers within its structure.
- The persistence of user data is not guaranteed due to the storage of metadata in a database. If the organization fails to manage this properly, it can result in the loss of information, leading to significant damages.
- Web3Auth is not yet widely adopted by dapps, so users may not be able to use their Web3Auth keys on all of the dapps that they want to use. This is because the platform is still relatively new, and it is not yet as widely integrated with dapps as other platforms, such as MetaMask.
- To successfully execute projects, various infrastructure components must be operational, with particular emphasis on the node's involvement in registering and creating encryption keys. Scaling out can result in significant infrastructure costs.

The system advantages:

- By utilizing smart contracts and blockchain technology, user data can be stored and secured in a decentralized manner, guaranteeing its durability and security as long as the blockchain network remains operational.
- By transferring the responsibility of constructing encKey and assignKey to the smart contract, the need for implementing intricate nodes to handle consensus between nodes, such as Web3Auth, is eliminated.

- The system's complexity decreased, resulting in significant reductions in infrastructure and debugging costs.

6.1 Conclusion

In conclusion, the system being discussed is an innovative and efficient solution for addressing identity and authorization challenges in the Web3 domain. This system has the potential to revolutionize online authentication and access by incorporating advanced features and functionalities. This solution utilizes advanced technologies such as blockchain, social login, and user-controlled encryption keys. It also incorporates robust security measures like smart contracts, Shamir's secret sharing, and the Pedersen DKG protocol. These features provide a seamless and secure user experience, enabling individuals to confidently engage with the Web3 ecosystem.

The system's user-friendly interface facilitates convenient navigation through the authentication and authorization processes, making it a notable strength. The incorporation of social login enables a smooth and recognizable authentication process, minimizing obstacles and improving user acceptance. In addition, user-controlled encryption keys offer individuals increased control and ownership of their private data, effectively addressing the escalating concerns surrounding data privacy and security.

Moreover, the system's capacity to streamline and automate the authorization process greatly improves efficiency and convenience. Virtualization technology enables the establishment of a uniform and stable deployment environment, facilitating the configuration and scalability of applications as required. The utilization of Express.js as the backend and JSON-RPC for communication enhances the system's robustness and performance, facilitating seamless and dependable interactions among various components.

The system's capacity to tackle the urgent issues of identity and authorization in the Web3 domain renders it a valuable asset for individuals, developers, and businesses. Decentralized applications are being increasingly adopted and utilized across various industries, leading to innovation and growth within the Web3 ecosystem. The Web3 landscape is constantly changing, and the system is prepared to adapt and provide secure and efficient solutions for identity and authorization in the decentralized digital world.

6.2 Future work

The system's beta version shows promise by offering support for Google Auth0 and web browsers. The development has promising prospects for the system, as there are plans to enhance its compatibility and extend its reach. The roadmap entails incorporating support for additional reputable authentication providers, including Facebook, Apple, Reddit, and others. The system seeks to expand its user base and provide users with a greater variety of authentication options by integrating with well-known authentication services. This integration allows for seamless identity verification.

Additionally, there are plans to improve the accessibility of the system by expanding its compatibility to various platforms, such as mobile devices and iPads. This expansion will enhance user interaction by accommodating various devices, thereby fostering inclusivity and improving user experience. The adoption of mobile platforms is essential for accommodating the growing number of users accessing the Web3 ecosystem via smartphones and tablets.

In the context of cross-device support, the system currently does not offer the capability to share the private key across multiple devices. However, this feature has been identified as a critical target for future development and enhancement of the social login device. The ability to seamlessly and securely share the private key across different devices is crucial to providing users with a consistent and flexible experience.

REFERENCE

[1] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, Nov. 1979.

[2] M. Abadi *et al.*, "Distributed key generation in the wild: A practice-oriented study," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2016, pp. 257–269.

[3] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, 2008. [Online]. Available: `https://bitcoin.org/bitcoin.pdf` (visited on 06/19/2023).

[4] V. Buterin and G. Wood, *Ethereum: A next-generation smart contract and decentralized application platform*, 2014. [Online]. Available: `https://ethereum.org/whitepaper/` (visited on 06/19/2023).

[5] A. M. Antonopoulos, *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. O'Reilly Media, 2014.

[6] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology - CRYPTO'87*, Springer, 1987, pp. 369–378. DOI: `10.1007/3-540-48184-2_32`.

[7] Bitcoin BIP-0039 Contributors, *Bip39: Mnemonic code for generating deterministic keys*, Bitcoin Improvement Proposal, Accessed on 2023-06-19, 2013. [Online]. Available: `https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki`.

[8] MetaMask, *Metamask: Ethereum wallet and gateway to blockchain apps*, Website, 2023. [Online]. Available: `https://metamask.io/` (visited on 06/19/2023).

[9] R. L. Burden and J. D. Faires, *Numerical Analysis*, 10th. Boston, MA: Cengage Learning, 2015, ISBN: 978-1-305-27108-9.

[10] T. P. Pedersen, "A threshold cryptosystem without a trusted party," in *Advances in Cryptology — EUROCRYPT'91*, Springer, 1991, pp. 522–526.

[11] N. Szabo, "Formalizing and securing relationships on public networks," in *First Monday*, vol. 2, University of Illinois at Chicago Library, 1997. DOI: `10.5210/fm.v2i9.548`. [Online]. Available: `http://firstmonday.org/ojs/index.php/fm/article/view/548/469`.

[12] C. van der Veen, *DeFi and the Future of Finance: A Comprehensive Guide to Decentralized Finance*. Packt Publishing, 2021, ISBN: 978-1801073687.

[13] M. Swan, *Token Economy: How the Web3 Reinvents Value Exchange*. 2018.

[14] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," 2014.

[15] Internet Engineering Task Force, *Oauth 2.0*, `https://tools.ietf.org/html/rfc6749`, Accessed: June 19, 2023, 2012.

[16] T. O'Reilly, *What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software*. O'Reilly Media, 2005.

[17] *Express.js*, `https://expressjs.com/`, Accessed: 2023-06-19.

[18] *JSON-RPC*, `https://www.jsonrpc.org/`, Accessed: 2023-06-19.

[19] *MySQL*, `https://www.mysql.com/`, Accessed: 2023-06-19.

[20] C. GmbH, *Cosmwasm: A secure and efficient smart contract platform for cosmos*, `https://www.cosmwasm.com/`, Accessed: 2023-06-19, 2021.

[21] C. Developers, *Cosmos sdk: A modular framework for building blockchain applications*, `https://cosmos.network/docs/tutorial/`, Accessed: 2023-06-19, 2021.

[22] D. Nock, *Autocannon*, https://github.com/mcollina/autocannon, version 7.11.0, Software package, GitHub.