User privacy in DAOs

Design Concept

Ignas Apšega

3620557

Version Control

Version	Date of change	Change description
0.1	30-10-2022	Initial draft
0.2	31-10-2022	C4 model diagrams
0.3	09-12-2022	1. Research questions. 2. Component diagrams.
Final Version	24-12-2022	Refactored diagrams, added conclusion

Glossary

Term	Explanation
1. Decentralized Autonomous Organisation (DAO)	DAO, or Decentralised Autonomous Organisation, is an organization without any hierarchy among its members. A DAO also has a set of “rules” on how to manage its treasury (DAO’s money) and how to handle votings, which are essential for making decisions, and a DAO usually has its own Token (Cryptocurrency, in this case). All these aspects are described inside a DAO smart contract.
2. Decentralized Application (dApp)	A decentralized application (dapp) is an application built on a decentralized network that combines a smart contract and a front-end user interface. On Ethereum, smart contracts are accessible and transparent – like open APIs – so your dapp can even include a smart contract that someone else has written. (“Introduction to dapps | ethereum.org”)
3. Layer 2	A layer 2 refers to any off-chain network, system, or technology built on top of a blockchain (commonly known as a layer-1 network) that helps extend the capabilities of the underlying base layer network. (“What Is a Layer 2?”)
4. Layer 1	Layer 1 refers to a base network, such as Bitcoin, BNB Chain, or Ethereum, and its underlying infrastructure. Layer-1 blockchains can validate and finalize transactions without the need for another network. Making improvements to the scalability of layer-1 networks is difficult, as we’ve seen with Bitcoin. As a solution,

	developers create layer-2 protocols that rely on the layer-1 network for security and consensus. Bitcoin's Lightning Network is one example of a layer-2 protocol. It allows users to make transactions freely before recording them into the main chain. (“What Is Layer 1 in Blockchain?”)
5. Zk-rollup	Runs computation off-chain and submits a validity proof to the chain. (“Scaling | ethereum.org”)
6. Optimistic rollup	Assumes transactions are valid by default and only runs computation, via a fraud-proof, in the event of a challenge. (“Scaling | ethereum.org”)
7. UML diagram	The Unified Modeling Language is a general-purpose, developmental modeling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system. (“Unified Modeling Language”)

Table of Contents

1. Introduction

1.1 Purpose of the document

This document is part of the User Privacy in DAOs[1] research assignment. It will showcase possible software architecture design for the decentralized application (dApp)[2]. In addition, this application will show how Zero-Knowledge technology can be implemented in this dApp.

1.2 Context

The main context of this dApp is that it should follow WEB3[3] rules thoroughly. Which includes using blockchain and being decentralized as much as possible. Also, it should implement Zero-Knowledge proof technology to bring privacy.

2. Research

2.1 Research Questions

This document was made to help visualize the possible architecture of the application. To help with research and design following sub-questions were followed:

1. **What tools are there for improving privacy for DAOs?**
2. **Could the solutions found be implemented in the current projects of Byont?**

2.2 Research strategy

For this research mainly, the strategies **Library**, **Field**, and **Workshop** were chosen.

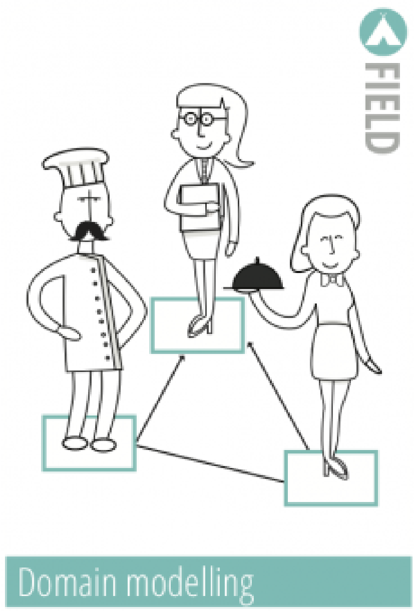
Library is done to explore what is already done and what guidelines and theories exist that could help.

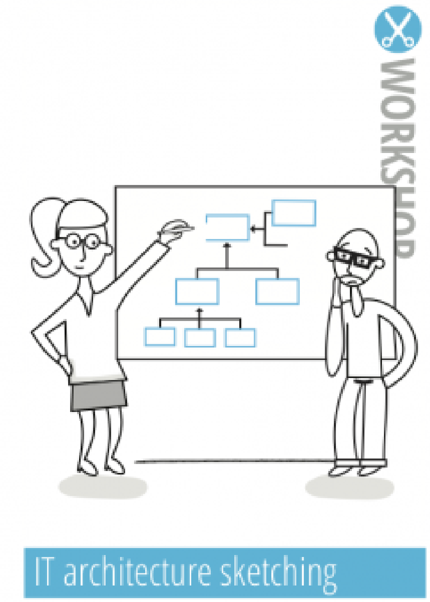
Field research is done to explore the application context.

Workshop research is done to explore opportunities. Prototyping, sketching, and co-creation activities are all ways to gain insights into what is possible and how things could work.

2.3 Research methods

Domain modeling - this research strategy was used to conclude in what context the application or proof of concept should be. Which leads to the conclusion of WEB3/Blockchain context.





IT architecture sketching - this research methodology was chosen to design the possible architecture of the application

Design pattern research was used to research well-known design patterns for WEB3/Blockchain applications.

3. Research Results

3.1 Web 2.0 vs Web 3.0

To understand Web 3.0 better, a comparison has to be made with Web 2.0:
Unlike Web 2.0 applications, Web 3.0 removes the middleman. There is no centralized web server where the backend logic resides, and no centralized database stores the application state.

Instead, blockchain technology is leveraged to build apps on a decentralized state machine that is maintained by anonymous nodes on the internet.

“State machine” means a machine that maintains a given program state and future states. Thus, blockchains are state machines that are instantiated with a genesis state and have very strict rules, such as consensus, that define how that state can transition.

To continue, no single entity controls this decentralized state machine - everyone in the network collectively maintains it.

Compared to how the backend is controlled in WEB2, in Web 3.0 a person can write smart contracts that define the application's logic and deploy it onto the decentralized state machine. This means everyone who wants to build a blockchain application deploys their code on this shared state machine. However, when frontends are compared, it stays pretty much the same in Web 3.0.

The possible architecture of the Web 3.0 application could be:

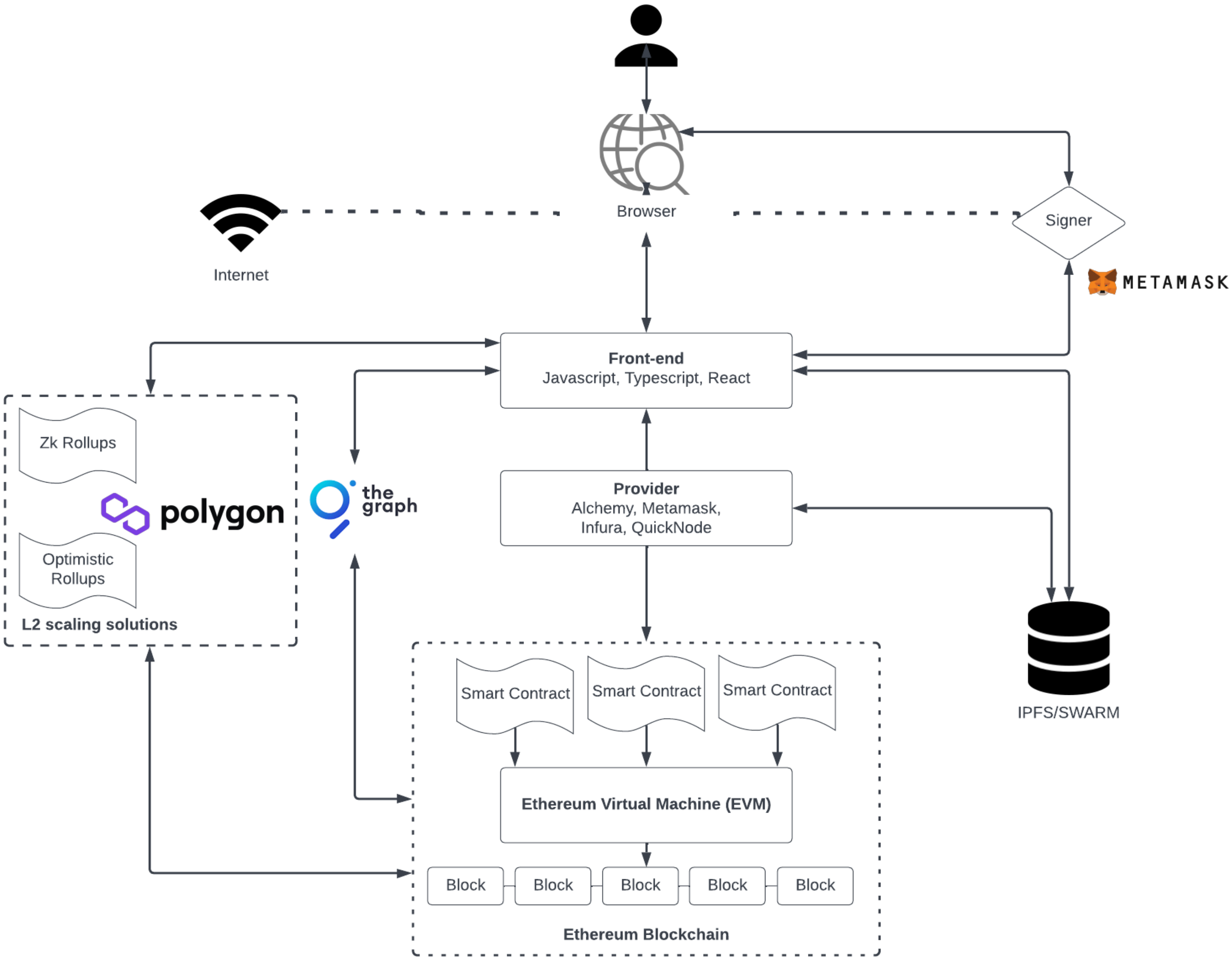


Figure 1. WEB 3.0 Architecture on Ethereum

3.2 Explanation

Front-end - It defines the UI logic. It also communicates with the application logic defined in smart contracts. Usually, in Web 3.0 front-end stack usually consists of Javascript, Typescript, and React for the logic.

Provider - Web3 provider allows your application to communicate with a Blockchain\Blockchain Node.

Signer - The most common Signers are Wallet, which knows its private key and can execute any operations. Once the connection to the blockchain through a provider is done, you can read the state stored on the blockchain. However, if you want to write to the state, one more thing must be done before you can submit the transaction to the blockchain, which is“sign” the transaction using your private key.

For example, a dApp lets users retrieve or publish blog posts to the blockchain. Also, let's assume there is a button on the front end that allows anyone to query for the written blog posts. To continue, reading from the blockchain does not require a user to sign a transaction.

But, when a user wants to publish a new post onto the blockchain, dApp would ask the user to “sign” the transaction using their private key — only then would the dApp pass on the transaction to the blockchain. Otherwise, the nodes of the blockchain would not accept the transaction.

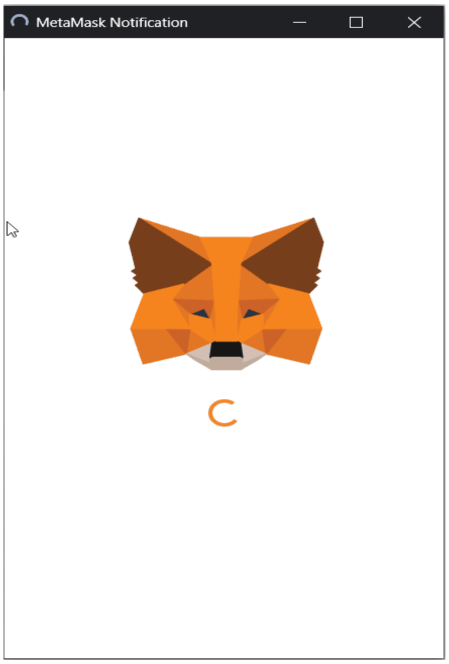


*Figure 2. Connecting and Signing on Metamask walle*t

Ethereum blockchain - It is designed to be a state machine that anyone in the world can access and write to. As a result, this machine is not owned by any single entity — but collectively by everyone in the network. Data can only be written to the Ethereum blockchain — however, you can not update existing data.

Smart contracts are programs that run on the blockchain and define the logic behind the state changes happening on the blockchain. Smart contracts can be written in programming languages, such as Solidity, Vyper, or Rust; it depends on the blockchain.

They are stored on the blockchain, and anyone can inspect the application logic of all smart contracts on the network.

Ethereum Virtual Machine (EVM) - It executes the logic defined in the smart contracts and processes the state changes that happen on this globally accessible state machine. The EVM does not understand high-level languages like Solidity and Vyper. Instead, it is compiled into bytecode, which the EVM can execute.

The Graph - helps to read data from the smart contracts on the blockchains. The Graph defines which smart contracts to index, which events and function call to listen to, and how to transform incoming events into entities that your frontend logic can use.
The Graph lets us query on-chain data with low latency by indexing blockchain data.

L2 scaling solutions are layer 2[3] scaling solutions for layer 1(e.g Ethereum). One popular scaling solution is Polygon. The way it works is that instead of executing transactions on the main blockchain - e.g Ethereum, Polygon has “sidechains” that process and execute transactions. A sidechain is a secondary blockchain that intersects with the main chain. Every then, the sidechain sends the gathering of its recent blocks back to the primary chain.

Other examples of L2 solutions are zkRollups[5] and Optimistic Rollups[6]. They work very similarly: Using a “rollup” smart contract, transactions are batched off-chain and then periodically committed to the main chain.

In summary: L2 solutions do transaction execution (the slow part) off-chain, with only the transaction data stored on-chain. This lets us scale the blockchain because we do not have to execute every single transaction on-chain. This also makes transactions cheaper and faster, and they can still communicate with the main Ethereum blockchain when necessary.

IPFS/SWARM - Decentralized off-chain storage solutions. These storages might be useful as a result storing everything on the blockchain can get expensive. And that is because you have to pay in cryptocurrency every time you write/post to the blockchain. That is because adding a state to the decentralized state machine increases the costs for nodes that are maintaining that state machine.
Asking users to pay extra for using your dApp every time their transaction requires adding a new state might not be the best user experience.

In addition, for full decentralization, the front end could be hosted via these storages.

3.3 Resolution

It is worth mentioning that the architecture displayed above is a possible solution but not necessarily the one. It is subject to change. For example:

1. **Change of blockchain** - instead of Ethereum, another blockchain could be used for this project. (e.g Avalanche, Solana)
2. **Another signer** - instead of Metamask, another signer could be used(e.g Phantom wallet).
3. **The Graph is optional** - There are providers who can also query the blockchain.
4. **IPFS/SWARM is optional** - It depends on how often the user will write to the blockchain. If it happens very often, you should store that data somewhere else - in decentralized storage. It is also worth mentioning that if you using a blockchain other than Ethereum (e.g Avalanche blockchain), the cost to write to the blockchain might be way cheaper.
5. **L2 scaling solutions are not necessary** - It is viable to use L2s if you want to build in the Ethereum ecosystem and save for writing to the blockchain. Moreover, you might want to use L2 to have everything on the blockchain transparent. However, if you do not, you can simply use decentralized storage IPFS/SWARM to save on writing to the blockchain and storing that data.

4. Possible Solution

4.1 C4 diagrams

C4 diagram is a software concept that is accepted worldwide. With a correctly built C4 diagram, developers can re-create the initial application. It consists of 4 parts that illustrate an application architecture on different levels: Context (C1), Container (C2), Component (C3), and Code (C4). With every next level, the diagram shows a more in-depth view of the software architecture, and the last level shows the code itself, usually using a UML class diagram.
A C4 diagram that was built for the application currently shows only the first three levels, as the coding has not been started yet. However, it is worth mentioning that this diagram may change during the development process, which also means that the last level will be added later. (“The C4 model for visualizing software architecture”)

4.2 Context diagram (C1)

The Context diagram focuses mainly on people that are going to use this application and separate independent software systems that are used by our system, but the ones that we can not control.

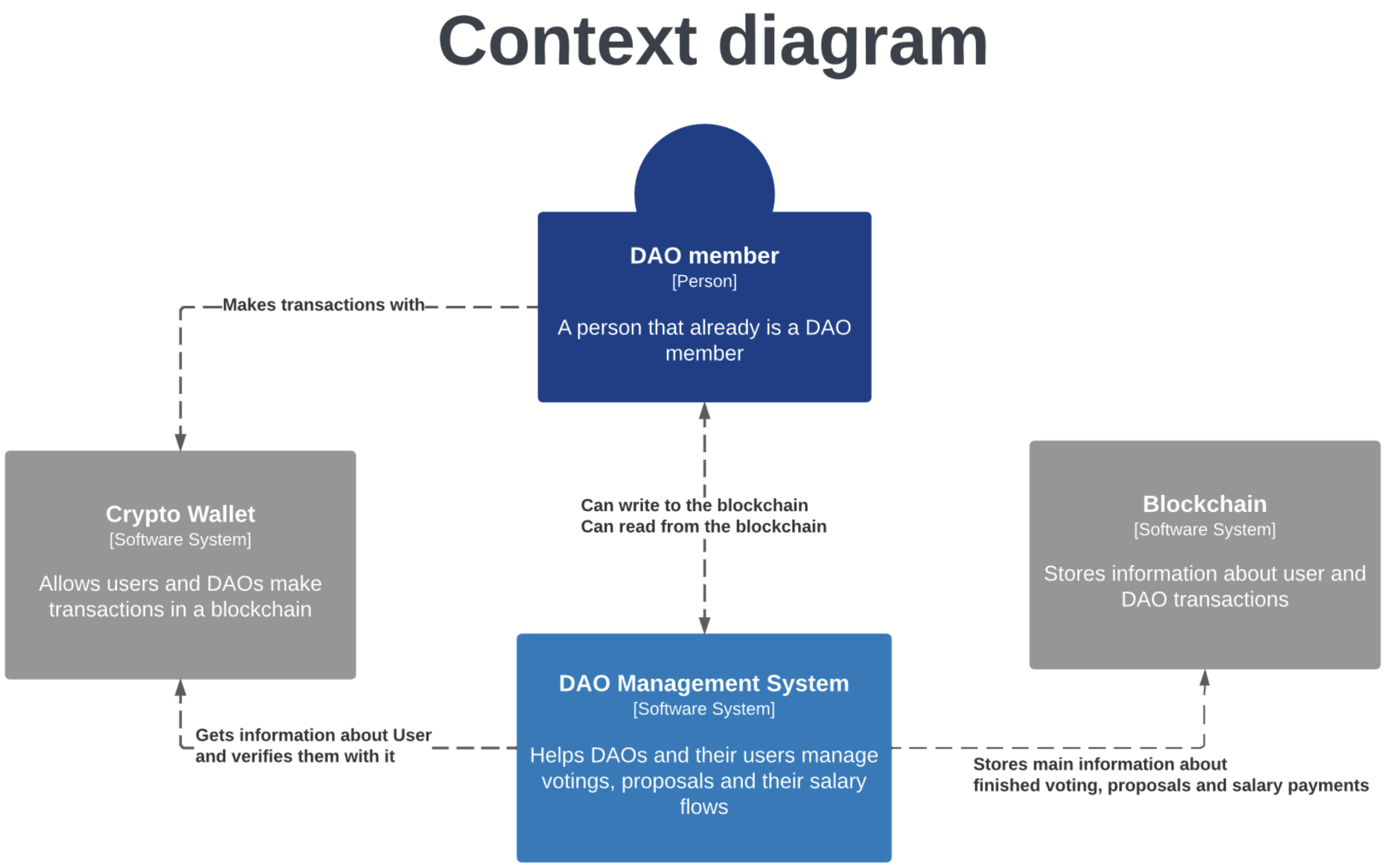

Context diagram
DAO member
[Person]
A person that already is a DAO member
Makes transactions with
Crypto Wallet
[Software System]
Allows users and DAOs make transactions in a blockchain
Can write to the blockchain
Can read from the blockchain
Blockchain
[Software System]
Stores information about user and DAO transactions
Gets information about User and verifies them with it
DAO Management System
[Software System]
Helps DAOs and their users manage votings, proposals and their salary flows
Stores main information about finished voting, proposals and salary payments

Figure 3. Context diagram (C1)

4.3 Description

Here, you can see a ***DAO member*** who can take action on the DAO management system. Those actions are: making proposals, voting, and receiving the salary. Every user will need to have a ***Crypto Wallet*** as this is necessary to sign transactions. Then the DAO management system through smart contracts will upload data to the ***Blockchain***.

4.4 Container diagram (C2)

The Container diagram goes a bit deeper and shows the high-level shape of the software architecture and how responsibilities are distributed across it.

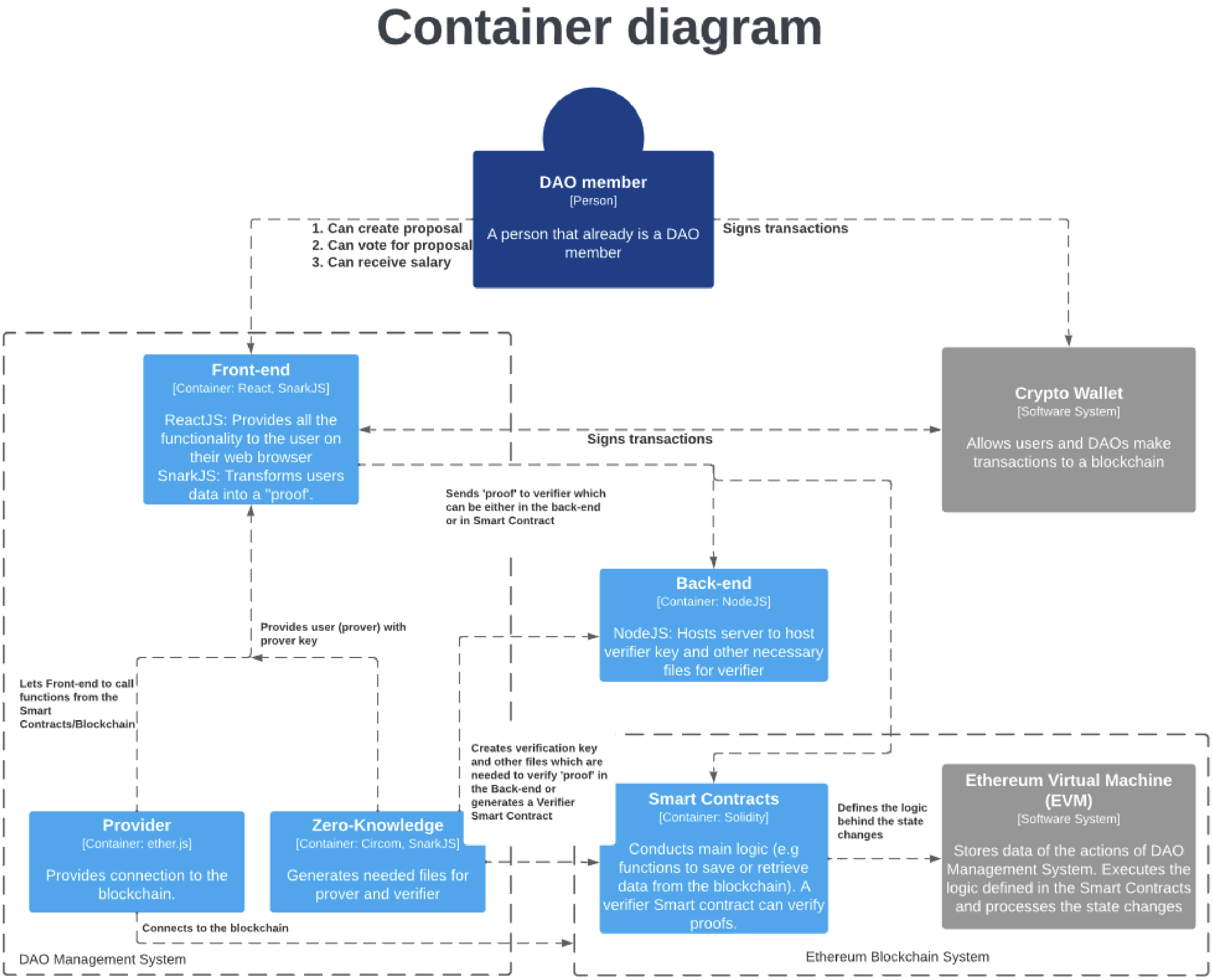

Container diagram
DAO member
[Person]
A person that already is a DAO member
1. Can create proposal
2. Can vote for proposal
3. Can receive salary
Signs transactions
Front-end
[Container: React, SnarkJS]
ReactJS: Provides all the functionality to the user on their web browser
SnarkJS: Transforms users data into a "proof".
Signs transactions
Crypto Wallet
[Software System]
Allows users and DAOs make transactions to a blockchain
Sends 'proof' to verifier which can be either in the back-end or in Smart Contract
Back-end
[Container: NodeJS]
NodeJS: Hosts server to host verifier key and other necessary files for verifier
Provides user (prover) with prover key
Lets Front-end to call functions from the Smart Contracts/Blockchain
Creates verification key and other files which are needed to verify 'proof' in the Back-end or generates a Verifier Smart Contract
Provider
[Container: ether.js]
Provides connection to the blockchain.
Zero-Knowledge
[Container: Circom, SnarkJS]
Generates needed files for prover and verifier
Smart Contracts
[Container: Solidity]
Conducts main logic (e.g functions to save or retrieve data from the blockchain). A verifier Smart contract can verify proofs.
Defines the logic behind the state changes
Ethereum Virtual Machine (EVM)
[Software System]
Stores data of the actions of DAO Management System. Executes the logic defined in the Smart Contracts and processes the state changes
Connects to the blockchain
DAO Management System
Ethereum Blockchain System

Figure 4. Container diagram (C2)

4.5 Description

DAO members will interact with the ***DAO Management System*** through front-end made in ReactJs. This front-end with the help of Web 3.0 provider(ether.js) will be able to store and retrieve data from the blockchain. Each time an user tries to store data to the blockchain a transaction has to be signed via ***Crypto Wallet.*** In addition, ***Provider*** will serves as an anonymous connection to the ethereum network, checking state and sending transactions. To continue, smart contracts will define the logic of the application and will be deployed to the **Ethereum Virtual Machine(EVM) - Ethereum blockchain.** Moreover, **Zero-Knowledge** container can transform entered data in a **'proof'** format which is encrypted and brings more privacy. Then, this **'proof'** can be verified via verifier - **Special Smart Contract** OR **Back-end Container** that indeed the entered data is correct without disclosing the data.

4.6 Component diagram - Front-end (C3)

The Component diagram shows how a container is made up of a number of "components", what each of those components is, their responsibilities, and the technology/implementation details.

4.7 Description

Front-end container could be made mostly out of two components **Data Collector Controller** and **Prover/Verifer Controller**

Data Collector Controller will be used to collect data from the front-end interface. Then with the help of **Prover/Verifier Controller,** this data can be converted into a 'proof' format which can be verified with the help of **the Back-end** or **Verifier Smart Contract.**

Component diagram (Front-end)

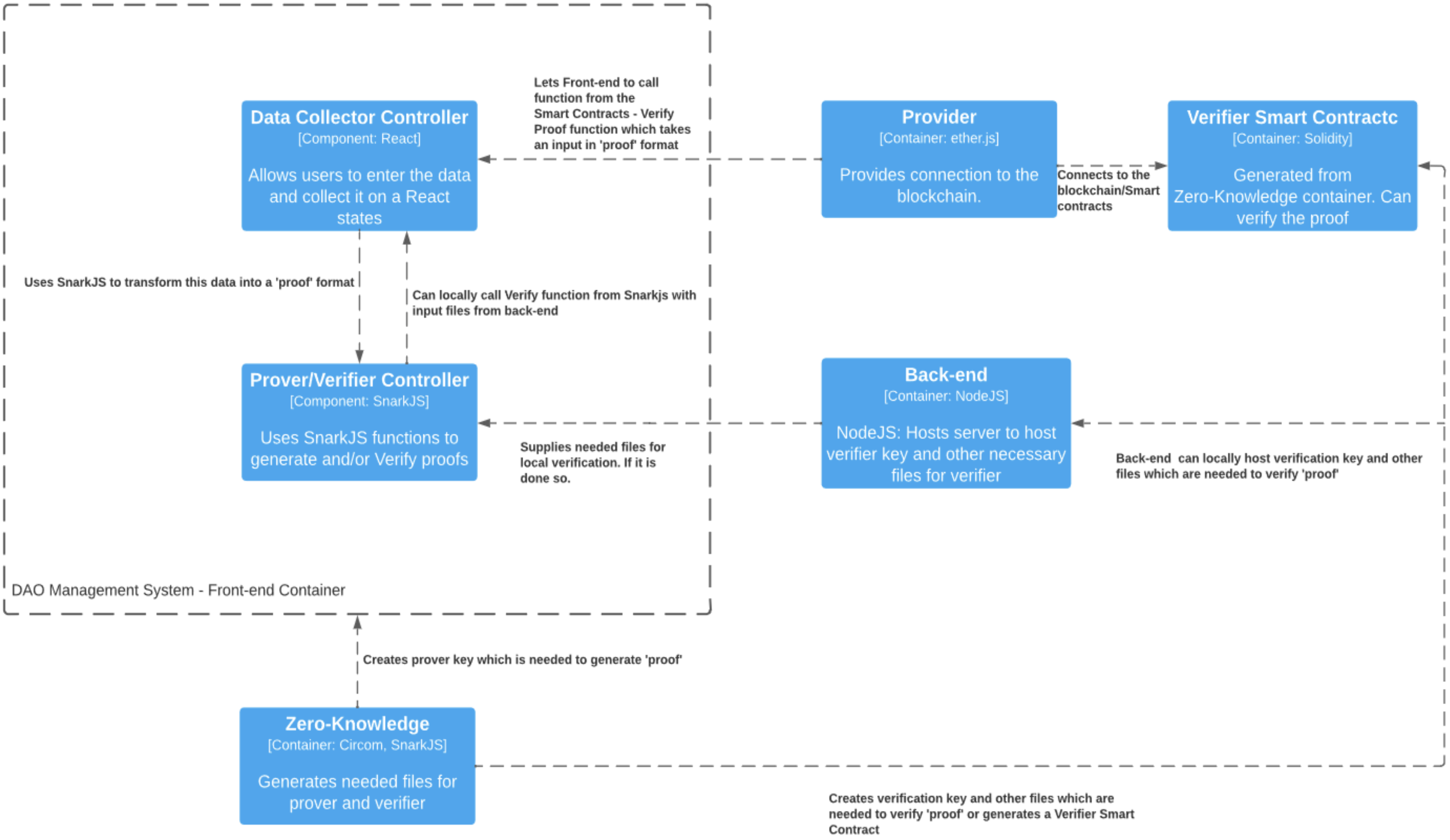
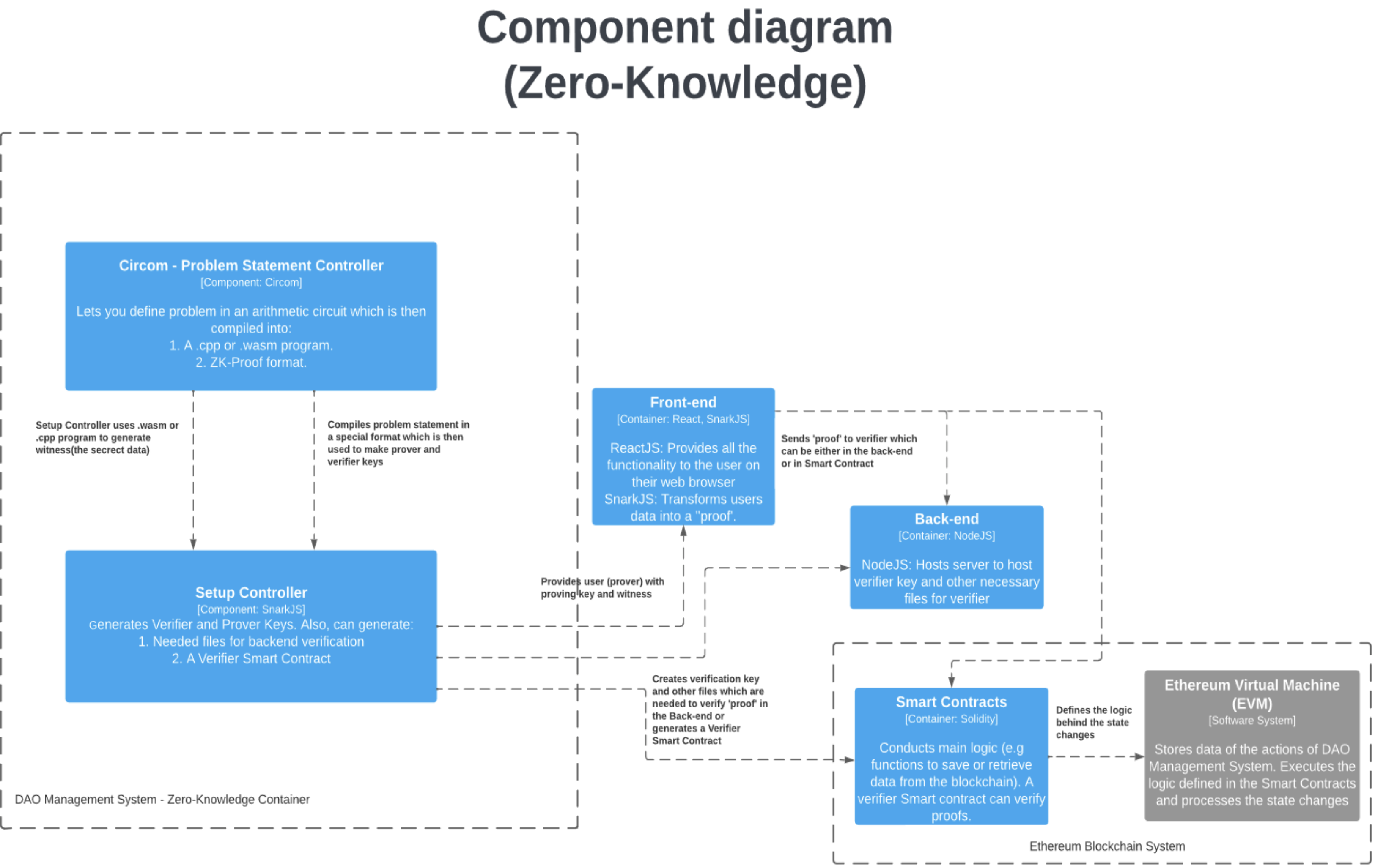

Lets Front-end to call function from the Smart Contracts - Verify Proof function which takes an input in 'proof' format
Data Collector Controller
[Component: React]
Allows users to enter the data and collect it on a React states
Provider
[Container: ether.js]
Provides connection to the blockchain.
Connects to the blockchain/Smart contracts
Verifier Smart Contractc
[Container: Solidity]
Generated from Zero-Knowledge container. Can verify the proof
Uses SnarkJS to transform this data into a 'proof' format
Can locally call Verify function from Snarkjs with input files from back-end
Prover/Verifier Controller
[Component: SnarkJS]
Uses SnarkJS functions to generate and/or Verify proofs
Supplies needed files for local verification. If it is done so.
Back-end
[Container: NodeJS]
NodeJS: Hosts server to host verifier key and other necessary files for verifier
Back-end can locally host verification key and other files which are needed to verify 'proof'
DAO Management System - Front-end Container
Creates prover key which is needed to generate 'proof'
Zero-Knowledge
[Container: Circom, SnarkJS]
Generates needed files for prover and verifier
Creates verification key and other files which are needed to verify 'proof' or generates a Verifier Smart Contract

Figure 5. Component diagram of Front-end (C3)

4.8 Component diagram - Zero-Knowledge (C3)

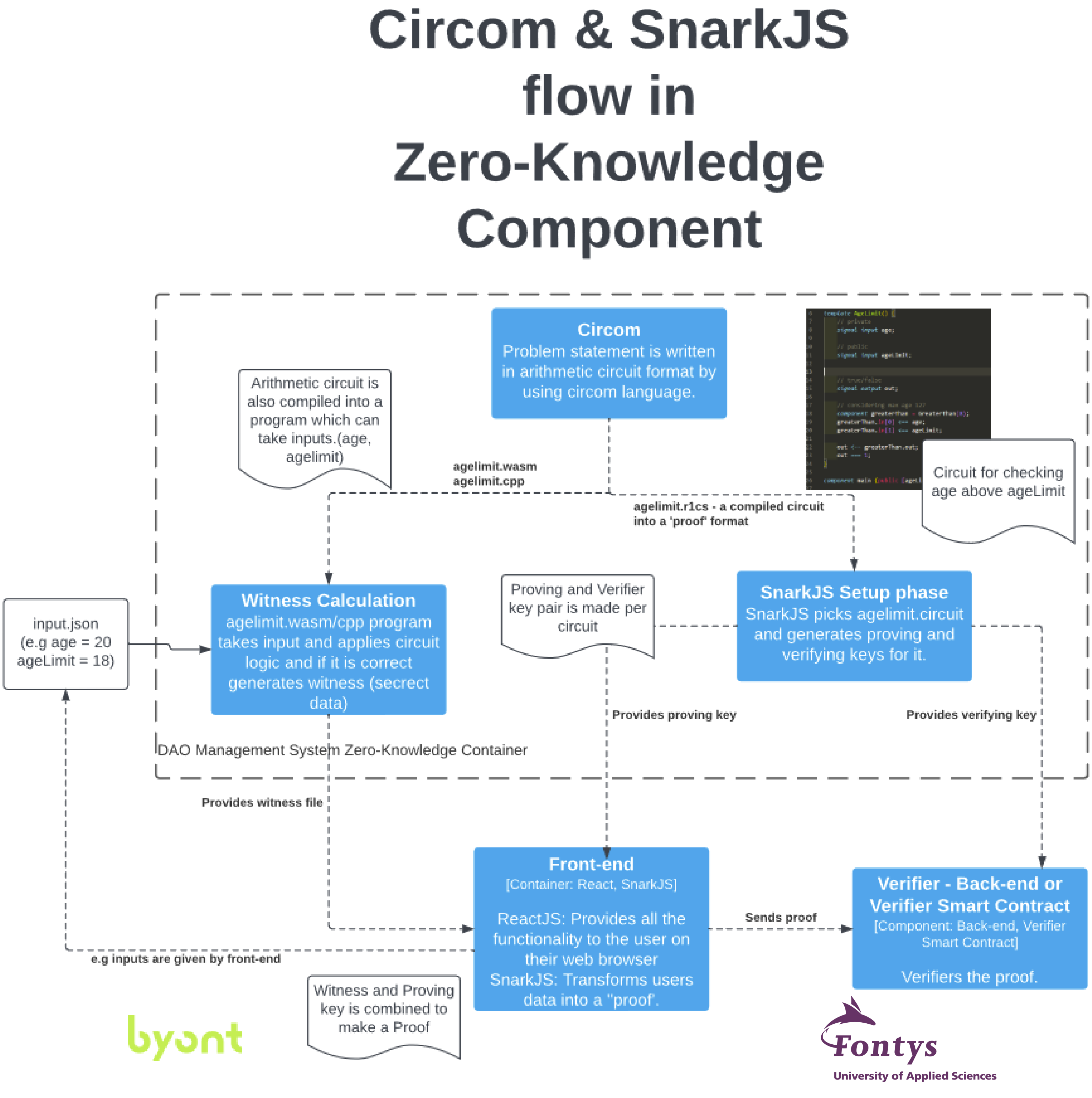

Component diagram
(Zero-Knowledge)
Circom - Problem Statement Controller
[Component: Circom]
Lets you define problem in an arithmetic circuit which is then compiled into:
1. A .cpp or .wasm program.
2. ZK-Proof format.
Setup Controller uses .wasm or .cpp program to generate witness(the secrect data)
Compiles problem statement in a special format which is then used to make prover and verifier keys
Setup Controller
[Component: SnarkJS]
Generates Verifier and Prover Keys. Also, can generate:
1. Needed files for backend verification
2. A Verifier Smart Contract
DAO Management System - Zero-Knowledge Container
Front-end
[Container: React, SnarkJS]
ReactJS: Provides all the functionality to the user on their web browser
SnarkJS: Transforms users data into a "proof".
Sends 'proof' to verifier which can be either in the back-end or in Smart Contract
Provides user (prover) with proving key and witness
Back-end
[Container: NodeJS]
NodeJS: Hosts server to host verifier key and other necessary files for verifier
Creates verification key and other files which are needed to verify 'proof' in the Back-end or generates a Verifier Smart Contract
Smart Contracts
[Container: Solidity]
Conducts main logic (e.g functions to save or retrieve data from the blockchain). A verifier Smart contract can verify proofs.
Defines the logic behind the state changes
Ethereum Virtual Machine (EVM)
[Software System]
Stores data of the actions of DAO Management System. Executes the logic defined in the Smart Contracts and processes the state changes
Ethereum Blockchain System

4.9 Description

Zero-Knowledge container is made out of two components **Circom - Problem Statement Controller** and **Setup Controller**

Circom - Problem Statement Controller is used to translate problem statement into a proof format. For example, it uses circom programming language to define the problem in an arithmetic circuit. Problem statement is basically logic that you are trying to prove e.g user is trying to prove that his age is above 18. To continue, Circom compiles this problem into a .wasm/.cpp program and into a special format which is then made into a proving key and verification key (this way, the program knows that you are trying to prove to the right verifier/circuit). The .wasm/.cpp program generates witness (secret data). It is done by supplying private input (age above 18) and public input (ageLimit 18). Then this witness together with the proving key is used on the Front-end to generate the proof which can be verified through the help of the **Back-end** or **Verifier Smart Contract.**

5. Zero-Knowledge flow



Circom & SnarkJS flow in Zero-Knowledge Component

```
template AgeLimit() {
    // private
    signal input age;

    // public
    signal input ageLimit;


    // true/false
    signal output out;

    // considering max age 127
    component greaterThan = GreaterThan(8);
    greaterThan.in[0] <== age;
    greaterThan.in[1] <== ageLimit;

    out <-- greaterThan.out;
    out === 1;
}

component main {public [ageLi
```

Circom
Problem statement is written in arithmetic circuit format by using circom language.
Arithmetic circuit is also compiled into a program which can take inputs.(age, agelimit)
Circuit for checking age above ageLimit
agelimit.wasm
agelimit.cpp
agelimit.r1cs - a compiled circuit into a 'proof' format
input.json
(e.g age = 20
ageLimit = 18)
Witness Calculation
agelimit.wasm/cpp program takes input and applies circuit logic and if it is correct generates witness (secrect data)
Proving and Verifier key pair is made per circuit
SnarkJS Setup phase
SnarkJS picks agelimit.circuit and generates proving and verifying keys for it.
Provides proving key
Provides verifying key
DAO Management System Zero-Knowledge Container
Provides witness file
e.g inputs are given by front-end
Front-end
[Container: React, SnarkJS]
ReactJS: Provides all the functionality to the user on their web browser
SnarkJS: Transforms users data into a "proof".
Sends proof
Verifier - Back-end or Verifier Smart Contract
[Component: Back-end, Verifier Smart Contract]
Verifiers the proof.
Witness and Proving key is combined to make a Proof
byont
Fontys
University of Applied Sciences

5. Conclusion

After researching possible solutions of how the WEB3 architecture of the application looks like. I came up with a solution of how the WEB3 application with Zero-Knowledge functionality could look like.

After discussion making the first Proof of Concept in this architecture. Further discussion with Byont was carried out to implement Zero-Knowledge functionality to their WEB3 template - an app that helps to bootstrap WEB3 applications.

Their WEB3 application function like the diagram shown in Figure 1. But it is worth mentioning that the Front-end is made in NextJS instead of ReactJS and the language for the Front-end is TypeScript instead of JavaScript.

Nonetheless, further progress and implementations will be made on Byonts WEB3 template.

Citation

“Bulletproofs | Stanford Applied Crypto Group.” *Applied Cryptography Group*, https://crypto.stanford.edu/bulletproofs/. Accessed 25 October 2022.

“The C4 model for visualising software architecture.” *The C4 model for visualising software architecture*, https://c4model.com/. Accessed 31 October 2022.

“Circom 2 Documentation.” *Circom 2 Documentation*, https://docs.circom.io/. Accessed 25 October 2022.

“dalek-cryptography/bulletproofs: A pure-Rust implementation of Bulletproofs using Ristretto.” *GitHub*, https://github.com/dalek-cryptography/bulletproofs. Accessed 25 October 2022.

“Domain-specific language.” *Wikipedia*, https://en.wikipedia.org/wiki/Domain-specific_language. Accessed 25 October 2022.

“iden3/circom: zkSnark circuit compiler.” *GitHub*, https://github.com/iden3/circom. Accessed 25 October 2022.

Ouaddah, Aafaf. “Arithmetic Circuit.” *ScienceDirect*, 2019, https://www.sciencedirect.com/topics/engineering/arithmetic-circuit. Accessed 25 10 2022.

“Scaling | ethereum.org.” *Ethereum.org*, https://ethereum.org/en/developers/docs/scaling/. Accessed 30 October 2022.

“Unified Modeling Language.” *Wikipedia*, https://en.wikipedia.org/wiki/Unified_Modeling_Language. Accessed 13 December 2022.

“What Is a Layer 2?” *Chainlink Blog*, 19 July 2022, https://blog.chain.link/what-is-a-layer-2/. Accessed 30 October 2022.

“What Is Layer 1 in Blockchain?” *Binance Academy*, 22 February 2022, https://academy.binance.com/en/articles/what-is-layer-1-in-blockchain. Accessed 30 October 2022.