README.md

**IDEN3 Smart Contracts**

* State - State contract, where identity states are published
* Smt - library to manage Sparse Merkle Trees onchain

**Security Audits**

[Nethermind](https://nethermind.io/smart-contracts-audits/) has performed a security audit of our core smart contracts (State & Smt) and compiled a report on Apr 18, 2023.

Here it is: [NM\_0069\_POLYGON\_FINAL.pdf](https://iden3-circuits-bucket.s3.eu-west-1.amazonaws.com/audit_reports/NM_0069_POLYGON_FINAL.pdf)

**Deployment**

Uncomment networks object and updated blockchain provider **url** and **private key** in hardhat.config.js for the relevant network. Then run the deployment script:

npx hardhat run --network <your-network> scripts/deploy.js

**Run tests**

npx hardhat test

Run tests with gas statistics report:

REPORT\_GAS=true npx hardhat test

Run tests with gas statistics and costs report:

COINMARKETCAP\_KEY=<<your coinmarketcap key>> REPORT\_GAS=true npx hardhat test

**Other Hardhat commands**

npx hardhat accounts

npx hardhat compile

npx hardhat clean

npx hardhat node

npx hardhat help

**Publish contracts to npm**

cd contracts

npm publish

**License**

This repository is part of the iden3 project copyright 2023 0KIMS Association and published under GPL-3.0 license. Please check the LICENSE file for more details.

contracts/state/README.md

## State smart contract

The contract is aimed to be a part of a core protocol for self-sovereign identity system. Its purpose is to store the meta information about identities.

In particular, State contract stores the data of uint256 type for each identity, which is called "identity state" or state. The state represents a root of special cryptographic structure, which is out of scope of smart contract logic. But the only thing that smart contract should check is that the caller knows some private key, which public key is inside the structure. The overall check is hidden in the Zero-knowledge proof verifier smart contract attached to the State contract. The verifier contract was automatically generated by the Iden3 SnarkJS ZK library and was audited by the library creators.

Another data structure used by State is a Sparse Merkle Tree, which keeps the whole tree on chain as well as its full history. Each tree leaf consist of an index and a value. The indexes keep identity identifier and values keep states. The main purpose of the SMT is to provide inclusion and non-inclusion proofs of the states of identities into the SMT root. The tree processing logic is implemented by the Smt library contract linked to the state contract.

Take into account that the State contract and SMT library import Poseidon.sol contract, which is nothing more than a "placeholder" for auto generated EVM code. The code is responsible for the onchain Poseidon hash implementation. There is a helper method in the repo, which uses the circomlibjs Poseidon ABI and code generator.

Finally, the State contract is OwnableUpgradeable, which means that it works via the ERC1967 Proxy contract, that keeps all its state. Any further upgrades of the State contract should avoid any storage collisions.

contracts/state/State.sol

// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.16**;**

**import** **{**Ownable2StepUpgradeable**}** **from** "@openzeppelin/contracts-upgradeable/access/Ownable2StepUpgradeable.sol"**;**

**import** **{**IState**,** MAX\_SMT\_DEPTH**}** **from** "../interfaces/IState.sol"**;**

**import** **{**IStateTransitionVerifier**}** **from** "../interfaces/IStateTransitionVerifier.sol"**;**

**import** **{**SmtLib**}** **from** "../lib/SmtLib.sol"**;**

**import** **{**PoseidonUnit1L**}** **from** "../lib/Poseidon.sol"**;**

**import** **{**StateLib**}** **from** "../lib/StateLib.sol"**;**

**import** **{**GenesisUtils**}** **from** "../lib/GenesisUtils.sol"**;**

/// @title Set and get states for each identity

contract State is Ownable2StepUpgradeable**,** IState **{**

/\*\*

\* @dev Version of contract

\*/

string **public** constant VERSION **=** "2.2.0"**;**

// This empty reserved space is put in place to allow future versions

// of the State contract to inherit from other contracts without a risk of

// breaking the storage layout. This is necessary because the parent contracts in the

// future may introduce some storage variables, which are placed before the State

// contract's storage variables.

// (see https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable#storage-gaps)

// slither-disable-next-line shadowing-state

// slither-disable-next-line unused-state

uint256**[**500**]** **private** \_\_gap**;**

/\*\*

\* @dev Verifier address

\*/

IStateTransitionVerifier internal verifier**;**

/\*\*

\* @dev State data

\*/

StateLib**.**Data internal \_stateData**;**

/\*\*

\* @dev Global Identity State Tree (GIST) data

\*/

SmtLib**.**Data internal \_gistData**;**

/\*\*

\* @dev Default Id Type

\*/

bytes2 internal \_defaultIdType**;**

/\*\*

\* @dev Default Id Type initialized flag

\*/

bool internal \_defaultIdTypeInitialized**;**

using SmtLib **for** SmtLib**.**Data**;**

using StateLib **for** StateLib**.**Data**;**

/// @custom:oz-upgrades-unsafe-allow constructor

constructor**()** **{**

\_disableInitializers**();**

**}**

/\*\*

\* @dev Initialize the contract

\* **@param** verifierContractAddr Verifier address

\*/

**function** initialize**(**

IStateTransitionVerifier verifierContractAddr**,**

bytes2 defaultIdType

**)** **public** initializer **{**

verifier **=** verifierContractAddr**;**

\_setDefaultIdType**(**defaultIdType**);**

\_gistData**.**initialize**(**MAX\_SMT\_DEPTH**);**

\_\_Ownable\_init**();**

**}**

/\*\*

\* @dev Set ZKP verifier contract address

\* **@param** newVerifierAddr Verifier contract address

\*/

**function** setVerifier**(**address newVerifierAddr**)** external onlyOwner **{**

verifier **=** IStateTransitionVerifier**(**newVerifierAddr**);**

**}**

/\*\*

\* @dev Set defaultIdType external wrapper (only owner can call)

\* **@param** defaultIdType default id type

\*/

**function** setDefaultIdType**(**bytes2 defaultIdType**)** external onlyOwner **{**

\_setDefaultIdType**(**defaultIdType**);**

**}**

/\*\*

\* @dev Change the state of an identity (transit to the new state) with ZKP ownership check.

\* **@param** id Identity

\* **@param** oldState Previous identity state

\* **@param** newState New identity state

\* **@param** isOldStateGenesis Is the previous state genesis?

\* **@param** a ZKP proof field

\* **@param** b ZKP proof field

\* **@param** c ZKP proof field

\*/

**function** transitState**(**

uint256 id**,**

uint256 oldState**,**

uint256 newState**,**

bool isOldStateGenesis**,**

uint256**[**2**]** memory a**,**

uint256**[**2**][**2**]** memory b**,**

uint256**[**2**]** memory c

**)** **public** **{**

uint256**[**4**]** memory input **=** **[**id**,** oldState**,** newState**,** uint256**(**isOldStateGenesis **?** 1 **:** 0**)];**

**require(**

verifier**.**verifyProof**(**a**,** b**,** c**,** input**),**

"Zero-knowledge proof of state transition is not valid"

**);**

\_transitState**(**id**,** oldState**,** newState**,** isOldStateGenesis**);**

**}**

/\*\*

\* @dev Change the state of an identity (transit to the new state) with method-specific id ownership check.

\* **@param** id Identity

\* **@param** oldState Previous identity state

\* **@param** newState New identity state

\* **@param** isOldStateGenesis Is the previous state genesis?

\* **@param** methodId State transition method id

\* **@param** methodParams State transition method-specific params

\*/

**function** transitStateGeneric**(**

uint256 id**,**

uint256 oldState**,**

uint256 newState**,**

bool isOldStateGenesis**,**

uint256 methodId**,**

bytes calldata methodParams

**)** **public** **{**

**if** **(**methodId **==** 1**)** **{**

uint256 calcId **=** GenesisUtils**.**calcOnchainIdFromAddress**(**

**this.**getDefaultIdType**(),**

msg**.**sender

**);**

**require(**calcId **==** id**,** "msg.sender is not owner of the identity"**);**

**require(**methodParams**.**length **==** 0**,** "methodParams should be empty"**);**

\_transitState**(**id**,** oldState**,** newState**,** isOldStateGenesis**);**

**}** **else** **{**

revert**(**"Unknown state transition method id"**);**

**}**

**}**

/\*\*

\* @dev Get ZKP verifier contract address

\* **@return** verifier contract address

\*/

**function** getVerifier**()** external view returns **(**address**)** **{**

**return** address**(**verifier**);**

**}**

/\*\*

\* @dev Get defaultIdType

\* **@return** defaultIdType

\*/

**function** getDefaultIdType**()** **public** view returns **(**bytes2**)** **{**

**require(**\_defaultIdTypeInitialized**,** "Default Id Type is not initialized"**);**

**return** \_defaultIdType**;**

**}**

/\*\*

\* @dev Retrieve the last state info for a given identity

\* **@param** id identity

\* **@return** state info of the last committed state

\*/

**function** getStateInfoById**(**uint256 id**)** external view returns **(**IState**.**StateInfo memory**)** **{**

**return** \_stateEntryInfoAdapter**(**\_stateData**.**getStateInfoById**(**id**));**

**}**

/\*\*

\* @dev Retrieve states quantity for a given identity

\* **@param** id identity

\* **@return** states quantity

\*/

**function** getStateInfoHistoryLengthById**(**uint256 id**)** external view returns **(**uint256**)** **{**

**return** \_stateData**.**getStateInfoHistoryLengthById**(**id**);**

**}**

/\*\*

\* Retrieve state infos for a given identity

\* **@param** id identity

\* **@param** startIndex start index of the state history

\* **@param** length length of the state history

\* **@return** A list of state infos of the identity

\*/

**function** getStateInfoHistoryById**(**

uint256 id**,**

uint256 startIndex**,**

uint256 length

**)** external view returns **(**IState**.**StateInfo**[]** memory**)** **{**

StateLib**.**EntryInfo**[]** memory stateInfos **=** \_stateData**.**getStateInfoHistoryById**(**

id**,**

startIndex**,**

length

**);**

IState**.**StateInfo**[]** memory result **=** **new** IState**.**StateInfo**[](**stateInfos**.**length**);**

**for** **(**uint256 i **=** 0**;** i **<** stateInfos**.**length**;** i**++)** **{**

result**[**i**]** **=** \_stateEntryInfoAdapter**(**stateInfos**[**i**]);**

**}**

**return** result**;**

**}**

/\*\*

\* @dev Retrieve state information by id and state.

\* **@param** id An identity.

\* **@param** state A state.

\* **@return** The state info.

\*/

**function** getStateInfoByIdAndState**(**

uint256 id**,**

uint256 state

**)** external view returns **(**IState**.**StateInfo memory**)** **{**

**return** \_stateEntryInfoAdapter**(**\_stateData**.**getStateInfoByIdAndState**(**id**,** state**));**

**}**

/\*\*

\* @dev Retrieve GIST inclusion or non-inclusion proof for a given identity.

\* **@param** id Identity

\* **@return** The GIST inclusion or non-inclusion proof for the identity

\*/

**function** getGISTProof**(**uint256 id**)** external view returns **(**IState**.**GistProof memory**)** **{**

**return** \_smtProofAdapter**(**\_gistData**.**getProof**(**PoseidonUnit1L**.**poseidon**([**id**])));**

**}**

/\*\*

\* @dev Retrieve GIST inclusion or non-inclusion proof for a given identity for

\* some GIST root in the past.

\* **@param** id Identity

\* **@param** root GIST root

\* **@return** The GIST inclusion or non-inclusion proof for the identity

\*/

**function** getGISTProofByRoot**(**

uint256 id**,**

uint256 root

**)** external view returns **(**IState**.**GistProof memory**)** **{**

**return** \_smtProofAdapter**(**\_gistData**.**getProofByRoot**(**PoseidonUnit1L**.**poseidon**([**id**]),** root**));**

**}**

/\*\*

\* @dev Retrieve GIST inclusion or non-inclusion proof for a given identity

\* for GIST latest snapshot by the block number provided.

\* **@param** id Identity

\* **@param** blockNumber Blockchain block number

\* **@return** The GIST inclusion or non-inclusion proof for the identity

\*/

**function** getGISTProofByBlock**(**

uint256 id**,**

uint256 blockNumber

**)** external view returns **(**IState**.**GistProof memory**)** **{**

**return**

\_smtProofAdapter**(**\_gistData**.**getProofByBlock**(**PoseidonUnit1L**.**poseidon**([**id**]),** blockNumber**));**

**}**

/\*\*

\* @dev Retrieve GIST inclusion or non-inclusion proof for a given identity

\* for GIST latest snapshot by the blockchain timestamp provided.

\* **@param** id Identity

\* **@param** timestamp Blockchain timestamp

\* **@return** The GIST inclusion or non-inclusion proof for the identity

\*/

**function** getGISTProofByTime**(**

uint256 id**,**

uint256 timestamp

**)** external view returns **(**IState**.**GistProof memory**)** **{**

**return** \_smtProofAdapter**(**\_gistData**.**getProofByTime**(**PoseidonUnit1L**.**poseidon**([**id**]),** timestamp**));**

**}**

/\*\*

\* @dev Retrieve GIST latest root.

\* **@return** The latest GIST root

\*/

**function** getGISTRoot**()** external view returns **(**uint256**)** **{**

**return** \_gistData**.**getRoot**();**

**}**

/\*\*

\* @dev Retrieve the GIST root history.

\* **@param** start Start index in the root history

\* **@param** length Length of the root history

\* **@return** Array of GIST roots infos

\*/

**function** getGISTRootHistory**(**

uint256 start**,**

uint256 length

**)** external view returns **(**IState**.**GistRootInfo**[]** memory**)** **{**

SmtLib**.**RootEntryInfo**[]** memory rootInfos **=** \_gistData**.**getRootHistory**(**start**,** length**);**

IState**.**GistRootInfo**[]** memory result **=** **new** IState**.**GistRootInfo**[](**rootInfos**.**length**);**

**for** **(**uint256 i **=** 0**;** i **<** rootInfos**.**length**;** i**++)** **{**

result**[**i**]** **=** \_smtRootInfoAdapter**(**rootInfos**[**i**]);**

**}**

**return** result**;**

**}**

/\*\*

\* @dev Retrieve the length of the GIST root history.

\* **@return** The GIST root history length

\*/

**function** getGISTRootHistoryLength**()** external view returns **(**uint256**)** **{**

**return** \_gistData**.**rootEntries**.**length**;**

**}**

/\*\*

\* @dev Retrieve the specific GIST root information.

\* **@param** root GIST root.

\* **@return** The GIST root information.

\*/

**function** getGISTRootInfo**(**uint256 root**)** external view returns **(**IState**.**GistRootInfo memory**)** **{**

**return** \_smtRootInfoAdapter**(**\_gistData**.**getRootInfo**(**root**));**

**}**

/\*\*

\* @dev Retrieve the GIST root information, which is latest by the block provided.

\* **@param** blockNumber Blockchain block number

\* **@return** The GIST root info

\*/

**function** getGISTRootInfoByBlock**(**

uint256 blockNumber

**)** external view returns **(**IState**.**GistRootInfo memory**)** **{**

**return** \_smtRootInfoAdapter**(**\_gistData**.**getRootInfoByBlock**(**blockNumber**));**

**}**

/\*\*

\* @dev Retrieve the GIST root information, which is latest by the blockchain timestamp provided.

\* **@param** timestamp Blockchain timestamp

\* **@return** The GIST root info

\*/

**function** getGISTRootInfoByTime**(**

uint256 timestamp

**)** external view returns **(**IState**.**GistRootInfo memory**)** **{**

**return** \_smtRootInfoAdapter**(**\_gistData**.**getRootInfoByTime**(**timestamp**));**

**}**

/\*\*

\* @dev Check if identity exists.

\* **@param** id Identity

\* **@return** True if the identity exists

\*/

**function** idExists**(**uint256 id**)** **public** view returns **(**bool**)** **{**

**return** \_stateData**.**idExists**(**id**);**

**}**

/\*\*

\* @dev Check if state exists.

\* **@param** id Identity

\* **@param** state State

\* **@return** True if the state exists

\*/

**function** stateExists**(**uint256 id**,** uint256 state**)** **public** view returns **(**bool**)** **{**

**return** \_stateData**.**stateExists**(**id**,** state**);**

**}**

/\*\*

\* @dev Change the state of an identity (transit to the new state) with ZKP ownership check.

\* **@param** id Identity

\* **@param** oldState Previous identity state

\* **@param** newState New identity state

\* **@param** isOldStateGenesis Is the previous state genesis?

\*/

**function** \_transitState**(**

uint256 id**,**

uint256 oldState**,**

uint256 newState**,**

bool isOldStateGenesis

**)** internal **{**

**require(**id **!=** 0**,** "ID should not be zero"**);**

**require(**newState **!=** 0**,** "New state should not be zero"**);**

**require(!**stateExists**(**id**,** newState**),** "New state already exists"**);**

**if** **(**isOldStateGenesis**)** **{**

**require(!**idExists**(**id**),** "Old state is genesis but identity already exists"**);**

// Push old state to state entries, with zero timestamp and block

\_stateData**.**addGenesisState**(**id**,** oldState**);**

**}** **else** **{**

**require(**idExists**(**id**),** "Old state is not genesis but identity does not yet exist"**);**

StateLib**.**EntryInfo memory prevStateInfo **=** \_stateData**.**getStateInfoById**(**id**);**

**require(**

prevStateInfo**.**createdAtBlock **!=** block**.**number**,**

"No multiple set in the same block"

**);**

**require(**prevStateInfo**.**state **==** oldState**,** "Old state does not match the latest state"**);**

**}**

\_stateData**.**addState**(**id**,** newState**);**

\_gistData**.**addLeaf**(**PoseidonUnit1L**.**poseidon**([**id**]),** newState**);**

**}**

**function** \_smtProofAdapter**(**

SmtLib**.**Proof memory proof

**)** internal pure returns **(**IState**.**GistProof memory**)** **{**

// slither-disable-next-line uninitialized-local

uint256**[**MAX\_SMT\_DEPTH**]** memory siblings**;**

**for** **(**uint256 i **=** 0**;** i **<** MAX\_SMT\_DEPTH**;** i**++)** **{**

siblings**[**i**]** **=** proof**.**siblings**[**i**];**

**}**

IState**.**GistProof memory result **=** IState**.**GistProof**({**

root**:** proof**.**root**,**

existence**:** proof**.**existence**,**

siblings**:** siblings**,**

index**:** proof**.**index**,**

value**:** proof**.**value**,**

auxExistence**:** proof**.**auxExistence**,**

auxIndex**:** proof**.**auxIndex**,**

auxValue**:** proof**.**auxValue

**});**

**return** result**;**

**}**

**function** \_smtRootInfoAdapter**(**

SmtLib**.**RootEntryInfo memory rootInfo

**)** internal pure returns **(**IState**.**GistRootInfo memory**)** **{**

**return**

IState**.**GistRootInfo**({**

root**:** rootInfo**.**root**,**

replacedByRoot**:** rootInfo**.**replacedByRoot**,**

createdAtTimestamp**:** rootInfo**.**createdAtTimestamp**,**

replacedAtTimestamp**:** rootInfo**.**replacedAtTimestamp**,**

createdAtBlock**:** rootInfo**.**createdAtBlock**,**

replacedAtBlock**:** rootInfo**.**replacedAtBlock

**});**

**}**

**function** \_stateEntryInfoAdapter**(**

StateLib**.**EntryInfo memory sei

**)** internal pure returns **(**IState**.**StateInfo memory**)** **{**

**return**

IState**.**StateInfo**({**

id**:** sei**.**id**,**

state**:** sei**.**state**,**

replacedByState**:** sei**.**replacedByState**,**

createdAtTimestamp**:** sei**.**createdAtTimestamp**,**

replacedAtTimestamp**:** sei**.**replacedAtTimestamp**,**

createdAtBlock**:** sei**.**createdAtBlock**,**

replacedAtBlock**:** sei**.**replacedAtBlock

**});**

**}**

/\*\*

\* @dev Set defaultIdType internal setter

\* **@param** defaultIdType default id type

\*/

**function** \_setDefaultIdType**(**bytes2 defaultIdType**)** internal **{**

\_defaultIdType **=** defaultIdType**;**

\_defaultIdTypeInitialized **=** **true;**

**}**

**}**

contracts/verifiers/ZKPVerifier.sol

// SPDX-License-Identifier: MIT

pragma solidity 0.8.16**;**

**import** **{**ERC20**}** **from** "@openzeppelin/contracts/token/ERC20/ERC20.sol"**;**

**import** **{**Ownable**}** **from** "@openzeppelin/contracts/access/Ownable.sol"**;**

**import** **{**GenesisUtils**}** **from** "../lib/GenesisUtils.sol"**;**

**import** **{**PoseidonFacade**}** **from** "../lib/Poseidon.sol"**;**

**import** **{**ICircuitValidator**}** **from** "../interfaces/ICircuitValidator.sol"**;**

**import** **{**IZKPVerifier**}** **from** "../interfaces/IZKPVerifier.sol"**;**

contract ZKPVerifier is IZKPVerifier**,** Ownable **{**

// msg.sender-> ( requestID -> is proof given )

mapping**(**address **=>** mapping**(**uint64 **=>** bool**))** **public** proofs**;**

mapping**(**uint64 **=>** ICircuitValidator**.**CircuitQuery**)** **public** requestQueries**;**

mapping**(**uint64 **=>** ICircuitValidator**)** **public** requestValidators**;**

uint64**[]** internal \_supportedRequests**;**

**function** submitZKPResponse**(**

uint64 requestId**,**

uint256**[]** calldata inputs**,**

uint256**[**2**]** calldata a**,**

uint256**[**2**][**2**]** calldata b**,**

uint256**[**2**]** calldata c

**)** **public** override returns **(**bool**)** **{**

**require(**

requestValidators**[**requestId**]** **!=** ICircuitValidator**(**address**(**0**)),**

"validator is not set for this request id"

**);** // validator exists

**require(**requestQueries**[**requestId**].**queryHash **!=** 0**,** "query is not set for this request id"**);** // query exists

\_beforeProofSubmit**(**requestId**,** inputs**,** requestValidators**[**requestId**]);**

**require(**

requestValidators**[**requestId**].**verify**(**

inputs**,**

a**,**

b**,**

c**,**

requestQueries**[**requestId**].**queryHash

**),**

"proof response is not valid"

**);**

proofs**[**msg**.**sender**][**requestId**]** **=** **true;** // user provided a valid proof for request

\_afterProofSubmit**(**requestId**,** inputs**,** requestValidators**[**requestId**]);**

**return** **true;**

**}**

**function** getZKPRequest**(**

uint64 requestId

**)** **public** view override returns **(**ICircuitValidator**.**CircuitQuery memory**)** **{**

**return** requestQueries**[**requestId**];**

**}**

**function** setZKPRequest**(**

uint64 requestId**,**

ICircuitValidator validator**,**

uint256 schema**,**

uint256 claimPathKey**,**

uint256 operator**,**

uint256**[]** calldata value

**)** **public** override onlyOwner returns **(**bool**)** **{**

uint256 valueHash **=** PoseidonFacade**.**poseidonSponge**(**value**);**

// only merklized claims are supported (claimPathNotExists is false, slot index is set to 0 )

uint256 queryHash **=** PoseidonFacade**.**poseidon6**(**

**[**schema**,** 0**,** operator**,** claimPathKey**,** 0**,** valueHash**]**

**);**

**return**

setZKPRequestRaw**(**

requestId**,**

validator**,**

schema**,**

claimPathKey**,**

operator**,**

value**,**

queryHash

**);**

**}**

**function** setZKPRequestRaw**(**

uint64 requestId**,**

ICircuitValidator validator**,**

uint256 schema**,**

uint256 claimPathKey**,**

uint256 operator**,**

uint256**[]** calldata value**,**

uint256 queryHash

**)** **public** override onlyOwner returns **(**bool**)** **{**

**if** **(**requestValidators**[**requestId**]** **==** ICircuitValidator**(**address**(**0x00**)))** **{**

\_supportedRequests**.**push**(**requestId**);**

**}**

requestQueries**[**requestId**].**queryHash **=** queryHash**;**

requestQueries**[**requestId**].**operator **=** operator**;**

requestQueries**[**requestId**].**circuitId **=** validator**.**getCircuitId**();**

requestQueries**[**requestId**].**claimPathKey **=** claimPathKey**;**

requestQueries**[**requestId**].**schema **=** schema**;**

requestQueries**[**requestId**].**value **=** value**;**

requestValidators**[**requestId**]** **=** validator**;**

**return** **true;**

**}**

**function** getSupportedRequests**()** **public** view returns **(**uint64**[]** memory arr**)** **{**

**return** \_supportedRequests**;**

**}**

/\*\*

\* @dev Hook that is called before any proof response submit

\*/

**function** \_beforeProofSubmit**(**

uint64 requestId**,**

uint256**[]** memory inputs**,**

ICircuitValidator validator

**)** internal virtual **{}**

/\*\*

\* @dev Hook that is called after any proof response submit

\*/

**function** \_afterProofSubmit**(**

uint64 requestId**,**

uint256**[]** memory inputs**,**

ICircuitValidator validator

**)** internal virtual **{}**

**}**

contracts/validators/CredentialAtomicQueryValidator.sol

// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.16**;**

**import** **{**OwnableUpgradeable**}** **from** "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol"**;**

**import** **{**GenesisUtils**}** **from** "../lib/GenesisUtils.sol"**;**

**import** **{**ICircuitValidator**}** **from** "../interfaces/ICircuitValidator.sol"**;**

**import** **{**IVerifier**}** **from** "../interfaces/IVerifier.sol"**;**

**import** **{**IState**}** **from** "../interfaces/IState.sol"**;**

**abstract** contract CredentialAtomicQueryValidator is OwnableUpgradeable**,** ICircuitValidator **{**

IVerifier **public** verifier**;**

IState **public** state**;**

uint256 **public** revocationStateExpirationTime**;**

uint256 **public** proofGenerationExpirationTime**;**

**function** initialize**(**

address \_verifierContractAddr**,**

address \_stateContractAddr

**)** **public** initializer **{**

revocationStateExpirationTime **=** 1 hours**;**

proofGenerationExpirationTime **=** 1 hours**;**

verifier **=** IVerifier**(**\_verifierContractAddr**);**

state **=** IState**(**\_stateContractAddr**);**

\_\_Ownable\_init**();**

**}**

**function** setRevocationStateExpirationTime**(**uint256 expirationTime**)** **public** virtual onlyOwner **{**

revocationStateExpirationTime **=** expirationTime**;**

**}**

**function** setProofGenerationExpirationTime**(**uint256 expirationTime**)** **public** virtual onlyOwner **{**

proofGenerationExpirationTime **=** expirationTime**;**

**}**

**function** getCircuitId**()** external pure virtual returns **(**string memory id**);**

**function** getChallengeInputIndex**()** external pure virtual returns **(**uint256 index**);**

**function** verify**(**

uint256**[]** calldata inputs**,**

uint256**[**2**]** calldata a**,**

uint256**[**2**][**2**]** calldata b**,**

uint256**[**2**]** calldata c**,**

uint256 queryHash

**)** external view virtual returns **(**bool**)** **{**

// verify that zkp is valid

**require(**verifier**.**verifyProof**(**a**,** b**,** c**,** inputs**),** "Proof is not valid"**);**

//destrcut values from result array

uint256**[]** memory validationParams **=** \_getInputValidationParameters**(**inputs**);**

uint256 inputQueryHash **=** validationParams**[**0**];**

**require(**inputQueryHash **==** queryHash**,** "query hash does not match the requested one"**);**

uint256 gistRoot **=** validationParams**[**1**];**

\_checkGistRoot**(**gistRoot**);**

uint256 issuerId **=** validationParams**[**2**];**

uint256 issuerClaissuerClaimState **=** validationParams**[**3**];**

\_checkStateContractOrGenesis**(**issuerId**,** issuerClaissuerClaimState**);**

uint256 issuerClaimNonRevState **=** validationParams**[**4**];**

\_checkClaimNonRevState**(**issuerId**,** issuerClaimNonRevState**);**

uint256 proofGenerationTimestamp **=** validationParams**[**5**];**

\_checkProofGeneratedExpiration**(**proofGenerationTimestamp**);**

**return** **(true);**

**}**

**function** \_getInputValidationParameters**(**

uint256**[]** calldata inputs

**)** internal pure virtual returns **(**uint256**[]** memory**);**

**function** \_checkGistRoot**(**uint256 gistRoot**)** internal view **{**

IState**.**GistRootInfo memory rootInfo **=** state**.**getGISTRootInfo**(**gistRoot**);**

**require(**rootInfo**.**root **==** gistRoot**,** "Gist root state isn't in state contract"**);**

**}**

**function** \_checkStateContractOrGenesis**(**uint256 \_id**,** uint256 \_state**)** internal view **{**

bool isStateGenesis **=** GenesisUtils**.**isGenesisState**(**\_id**,** \_state**);**

**if** **(!**isStateGenesis**)** **{**

IState**.**StateInfo memory stateInfo **=** state**.**getStateInfoByIdAndState**(**\_id**,** \_state**);**

**require(**\_id **==** stateInfo**.**id**,** "state doesn't exist in state contract"**);**

**}**

**}**

**function** \_checkClaimNonRevState**(**uint256 \_id**,** uint256 \_claimNonRevState**)** internal view **{**

IState**.**StateInfo memory claimNonRevStateInfo **=** state**.**getStateInfoById**(**\_id**);**

**if** **(**claimNonRevStateInfo**.**state **==** 0**)** **{**

**require(**

GenesisUtils**.**isGenesisState**(**\_id**,** \_claimNonRevState**),**

"Non-Revocation state isn't in state contract and not genesis"

**);**

**}** **else** **{**

// The non-empty state is returned, and it's not equal to the state that the user has provided.

**if** **(**claimNonRevStateInfo**.**state **!=** \_claimNonRevState**)** **{**

// Get the time of the latest state and compare it to the transition time of state provided by the user.

IState**.**StateInfo memory claimNonRevLatestStateInfo **=** state**.**getStateInfoByIdAndState**(**

\_id**,**

\_claimNonRevState

**);**

**if** **(**claimNonRevLatestStateInfo**.**id **==** 0 **||** claimNonRevLatestStateInfo**.**id **!=** \_id**)** **{**

revert**(**"state in transition info contains invalid id"**);**

**}**

**if** **(**claimNonRevLatestStateInfo**.**replacedAtTimestamp **==** 0**)** **{**

revert**(**"Non-Latest state doesn't contain replacement information"**);**

**}**

**if** **(**

block**.**timestamp **-** claimNonRevLatestStateInfo**.**replacedAtTimestamp **>**

revocationStateExpirationTime

**)** **{**

revert**(**"Non-Revocation state of Issuer expired"**);**

**}**

**}**

**}**

**}**

**function** \_checkProofGeneratedExpiration**(**uint256 \_proofGenerationTimestamp**)** internal view **{**

**if** **(**block**.**timestamp **-** \_proofGenerationTimestamp **>** proofGenerationExpirationTime**)** **{**

revert**(**"Generated proof is outdated"**);**

**}**

**}**

**}**

contracts/validators/CredentialAtomicQuerySigValidator.sol

// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.16**;**

**import** **{**CredentialAtomicQueryValidator**}** **from** "./CredentialAtomicQueryValidator.sol"**;**

contract CredentialAtomicQuerySigValidator is CredentialAtomicQueryValidator **{**

string internal constant CIRCUIT\_ID **=** "credentialAtomicQuerySigV2OnChain"**;**

uint256 internal constant CHALLENGE\_INDEX **=** 5**;**

**function** getCircuitId**()** external pure override returns **(**string memory id**)** **{**

**return** CIRCUIT\_ID**;**

**}**

**function** getChallengeInputIndex**()** external pure override returns **(**uint256 index**)** **{**

**return** CHALLENGE\_INDEX**;**

**}**

**function** \_getInputValidationParameters**(**

uint256**[]** calldata inputs

**)** internal pure override returns **(**uint256**[]** memory**)** **{**

uint256**[]** memory params **=** **new** uint256**[](**6**);**

params**[**0**]** **=** inputs**[**2**];** // queryHash

params**[**1**]** **=** inputs**[**6**];** // gistRoot

params**[**2**]** **=** inputs**[**7**];** // issuerId

params**[**3**]** **=** inputs**[**3**];** // issuerClaimAuthState

params**[**4**]** **=** inputs**[**9**];** // issuerClaimNonRevState

params**[**5**]** **=** inputs**[**10**];** // timestamp

**return** params**;**

**}**

**}**

contracts/validators/CredentialAtomicQueryMTPValidator.sol

// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.16**;**

**import** **{**CredentialAtomicQueryValidator**}** **from** "./CredentialAtomicQueryValidator.sol"**;**

contract CredentialAtomicQueryMTPValidator is CredentialAtomicQueryValidator **{**

string internal constant CIRCUIT\_ID **=** "credentialAtomicQueryMTPV2OnChain"**;**

uint256 internal constant CHALLENGE\_INDEX **=** 4**;**

**function** getCircuitId**()** external pure override returns **(**string memory id**)** **{**

**return** CIRCUIT\_ID**;**

**}**

**function** getChallengeInputIndex**()** external pure override returns **(**uint256 index**)** **{**

**return** CHALLENGE\_INDEX**;**

**}**

**function** \_getInputValidationParameters**(**

uint256**[]** calldata inputs

**)** internal pure override returns **(**uint256**[]** memory**)** **{**

uint256**[]** memory params **=** **new** uint256**[](**6**);**

params**[**0**]** **=** inputs**[**2**];** // queryHash

params**[**1**]** **=** inputs**[**5**];** // gistRoot

params**[**2**]** **=** inputs**[**6**];** // issuerId

params**[**3**]** **=** inputs**[**7**];** // issuerClaimIdenState

params**[**4**]** **=** inputs**[**9**];** // issuerClaimNonRevState

params**[**5**]** **=** inputs**[**10**];** // timestamp

**return** params**;**

**}**

**}**

contracts/lib/verifierSigWrapper.sol

//

// Copyright 2017 Christian Reitwiessner

// Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

// The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

//

// 2019 OKIMS

// ported to solidity 0.6

// fixed linter warnings

// added requiere error messages

//

//

// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.16**;**

**import** "./verifierSig.sol"**;**

**import** "../interfaces/IVerifier.sol"**;**

contract VerifierSigWrapper is VerifierSig**,** IVerifier **{**

/// **@return** r bool true if proof is valid

**function** verifyProof**(**

uint256**[**2**]** memory a**,**

uint256**[**2**][**2**]** memory b**,**

uint256**[**2**]** memory c**,**

uint256**[]** memory input

**)** **public** view returns **(**bool r**)** **{**

// slither-disable-next-line uninitialized-local

Proof memory proof**;**

proof**.**A **=** Pairing**.**G1Point**(**a**[**0**],** a**[**1**]);**

proof**.**B **=** Pairing**.**G2Point**([**b**[**0**][**0**],** b**[**0**][**1**]],** **[**b**[**1**][**0**],** b**[**1**][**1**]]);**

proof**.**C **=** Pairing**.**G1Point**(**c**[**0**],** c**[**1**]);**

**if** **(**verify**(**input**,** proof**)** **==** 0**)** **{**

**return** **true;**

**}**

**return** **false;**

**}**

**}**

contracts/lib/verifierMTPWrapper.sol

//

// Copyright 2017 Christian Reitwiessner

// Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

// The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

//

// 2019 OKIMS

// ported to solidity 0.6

// fixed linter warnings

// added requiere error messages

//

//

// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.16**;**

**import** "./verifierMTP.sol"**;**

**import** "../interfaces/IVerifier.sol"**;**

contract VerifierMTPWrapper is VerifierMTP**,** IVerifier **{**

/// **@return** r bool true if proof is valid

**function** verifyProof**(**

uint256**[**2**]** memory a**,**

uint256**[**2**][**2**]** memory b**,**

uint256**[**2**]** memory c**,**

uint256**[]** memory input

**)** **public** view returns **(**bool r**)** **{**

// slither-disable-next-line uninitialized-local

Proof memory proof**;**

proof**.**A **=** Pairing**.**G1Point**(**a**[**0**],** a**[**1**]);**

proof**.**B **=** Pairing**.**G2Point**([**b**[**0**][**0**],** b**[**0**][**1**]],** **[**b**[**1**][**0**],** b**[**1**][**1**]]);**

proof**.**C **=** Pairing**.**G1Point**(**c**[**0**],** c**[**1**]);**

**if** **(**verify**(**input**,** proof**)** **==** 0**)** **{**

**return** **true;**

**}** **else** **{**

**return** **false;**

**}**

**}**

**}**

contracts/lib/Verifier.sol

//

// Copyright 2017 Christian Reitwiessner

// Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

// The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

//

// 2019 OKIMS

// ported to solidity 0.6

// fixed linter warnings

// added requiere error messages

//

//

// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.16**;**

**import** "../lib/Pairing.sol"**;**

**import** "../interfaces/IStateTransitionVerifier.sol"**;**

contract Verifier is IStateTransitionVerifier **{**

using Pairing **for** **\*;**

struct VerifyingKey **{**

Pairing**.**G1Point alfa1**;**

Pairing**.**G2Point beta2**;**

Pairing**.**G2Point gamma2**;**

Pairing**.**G2Point delta2**;**

Pairing**.**G1Point**[]** IC**;**

**}**

struct Proof **{**

Pairing**.**G1Point A**;**

Pairing**.**G2Point B**;**

Pairing**.**G1Point C**;**

**}**

**function** verifyingKey**()** internal pure returns **(**VerifyingKey memory vk**)** **{**

vk**.**alfa1 **=** Pairing**.**G1Point**(**

20491192805390485299153009773594534940189261866228447918068658471970481763042**,**

9383485363053290200918347156157836566562967994039712273449902621266178545958

**);**

vk**.**beta2 **=** Pairing**.**G2Point**(**

**[**

4252822878758300859123897981450591353533073413197771768651442665752259397132**,**

6375614351688725206403948262868962793625744043794305715222011528459656738731

**],**

**[**

21847035105528745403288232691147584728191162732299865338377159692350059136679**,**

10505242626370262277552901082094356697409835680220590971873171140371331206856

**]**

**);**

vk**.**gamma2 **=** Pairing**.**G2Point**(**

**[**

11559732032986387107991004021392285783925812861821192530917403151452391805634**,**

10857046999023057135944570762232829481370756359578518086990519993285655852781

**],**

**[**

4082367875863433681332203403145435568316851327593401208105741076214120093531**,**

8495653923123431417604973247489272438418190587263600148770280649306958101930

**]**

**);**

vk**.**delta2 **=** Pairing**.**G2Point**(**

**[**

4246152484702050277565132335408650010216666048103975186858037423667921011245**,**

11761106885383518720174451196687963724495127702612880995502231202411849421701

**],**

**[**

20662719780693521898375922787282175696841448037933826627867273008735335783602**,**

9540218714987219778576059617464635889429392349728954857252076100095683267633

**]**

**);**

vk**.**IC **=** **new** Pairing**.**G1Point**[](**5**);**

vk**.**IC**[**0**]** **=** Pairing**.**G1Point**(**

16043291973889324756617069487195476149512574727363051659112556958735977616725**,**

16864605224185193093062266789812233298859884301538621362226822022081041278677

**);**

vk**.**IC**[**1**]** **=** Pairing**.**G1Point**(**

15935621905201691307201070923038920580506689594547556653696264182846970978554**,**

20793947184131761785325026067954699416249353321530615459908048240252442935417

**);**

vk**.**IC**[**2**]** **=** Pairing**.**G1Point**(**

15873695673932800019757092006642463598109301274410205214955538808836281067900**,**

13581010826645089044340117513778871694012835043547906854734814490388643425494

**);**

vk**.**IC**[**3**]** **=** Pairing**.**G1Point**(**

436067793811322464859758359330968701378288169738014324837094148538366747065**,**

5184689509856778472522887232562113210294765146488556347841833551753176606959

**);**

vk**.**IC**[**4**]** **=** Pairing**.**G1Point**(**

1580946655352989990810599848244095954566838172532565943008224849077018394283**,**

8901953775389474246223858845884219088656635610469822712500097959042485592148

**);**

**}**

**function** verify**(**uint**[]** memory input**,** Proof memory proof**)** internal view returns **(**uint**)** **{**

uint256 snark\_scalar\_field **=** 21888242871839275222246405745257275088548364400416034343698204186575808495617**;**

VerifyingKey memory vk **=** verifyingKey**();**

**require(**input**.**length **+** 1 **==** vk**.**IC**.**length**,** "verifier-bad-input"**);**

// Compute the linear combination vk\_x

Pairing**.**G1Point memory vk\_x **=** Pairing**.**G1Point**(**0**,** 0**);**

**for** **(**uint i **=** 0**;** i **<** input**.**length**;** i**++)** **{**

**require(**input**[**i**]** **<** snark\_scalar\_field**,** "verifier-gte-snark-scalar-field"**);**

vk\_x **=** Pairing**.**addition**(**vk\_x**,** Pairing**.**scalar\_mul**(**vk**.**IC**[**i **+** 1**],** input**[**i**]));**

**}**

vk\_x **=** Pairing**.**addition**(**vk\_x**,** vk**.**IC**[**0**]);**

**if** **(**

**!**Pairing**.**pairingProd4**(**

Pairing**.**negate**(**proof**.**A**),**

proof**.**B**,**

vk**.**alfa1**,**

vk**.**beta2**,**

vk\_x**,**

vk**.**gamma2**,**

proof**.**C**,**

vk**.**delta2

**)**

**)** **return** 1**;**

**return** 0**;**

**}**

/// **@return** r bool true if proof is valid

**function** verifyProof**(**

uint**[**2**]** memory a**,**

uint**[**2**][**2**]** memory b**,**

uint**[**2**]** memory c**,**

uint**[**4**]** memory input

**)** **public** view returns **(**bool r**)** **{**

// slither-disable-next-line uninitialized-local

Proof memory proof**;**

proof**.**A **=** Pairing**.**G1Point**(**a**[**0**],** a**[**1**]);**

proof**.**B **=** Pairing**.**G2Point**([**b**[**0**][**0**],** b**[**0**][**1**]],** **[**b**[**1**][**0**],** b**[**1**][**1**]]);**

proof**.**C **=** Pairing**.**G1Point**(**c**[**0**],** c**[**1**]);**

uint**[]** memory inputValues **=** **new** uint**[](**input**.**length**);**

**for** **(**uint i **=** 0**;** i **<** input**.**length**;** i**++)** **{**

inputValues**[**i**]** **=** input**[**i**];**

**}**

**if** **(**verify**(**inputValues**,** proof**)** **==** 0**)** **{**

**return** **true;**

**}** **else** **{**

**return** **false;**

**}**

**}**

**}**

个人理解：

看到现在，对大部分主要的合约都看了，但还没有搞清楚这些合约有什么用，在什么场景下使用，参数来自于哪里，被谁调用的，整个流程还没有串起来。

继续看看test里的单元测试用例。

test里的单元测试用例都非常底层，和上层业务没有关系，看得出来，整个repository都是非常底层的合约，和业务没有关系。先暂且放一放。