Kairos: Zero-knowledge Casper Transaction Scaling

Design

Marijan Petricevic, Nick Van den Broeck, Mark Greenslade, Tom Sydney Kerckhove, Matthew Doty, Avi Dessauer, Jonas Pauli, Andrzej Bronski, Quinn Dougherty, Chloe Kever

Tuesday November 28, 2023

Contents

1. Architecture

To give the reader an idea of what a system defined in the previous sections might look like, this section proposes a possible architecture. The features and requirements described previously suggest two core components in the system. A CLI and an L2 node implementing the client-server pattern. The L2 node should not be monolithic but rather obey the principle of separation of concerns to allow for easier extensibility and replacement of components in the future. Therefore, the L2 node unifies various other components described in more detail in the following Section 1.1 and exposes them through a single API.

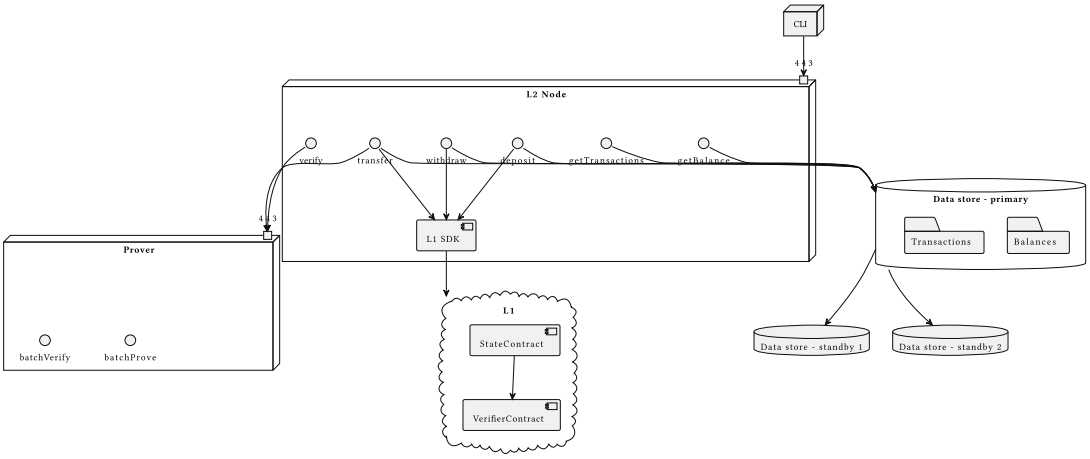

Figure 1: Components diagram

1.1. Architecture Components

1.1.1. Client CLI

The client CLI's user interface (UI) is comprised of commands allowing users to deposit, transfer, and withdraw funds allocated in their L2 account. Once a user executes any of the commands, the client delegates the bulk of the work to the L2 node (Section 1.1.2). Moreover, the client CLI can be used to verify past state changes and to query account balances and transfers.

1.1.2. L2 Node

As constrained in the requirements document, the L2 node is centralized. The detailed reasoning behind this decision, potential dangers, and our methods for dealing with these dangers are explained in [1].

The L2 node is the interface through which external clients (Section 1.1.1) can submit deposits, transfers, or withdrawals of funds. Moreover, it is connected to a

data store (Section 1.1.4) to persist the account balances, whose state representation[1] is maintained on-chain. State transitions of the account balances are verified and executed on-chain, requiring the node to create respective transactions on L1 using the L1′s software development kit (SDK). These transactions, in turn, call the respective endpoints of smart contracts described in Section 1.1.5 to do so. To execute batch transfers the node utilizes a proving system provided by the Prover service (Section 1.1.3). For deposits and withdrawals, the node creates according Merkle tree updates of the account balances [2].

1.1.3. Prover

The Prover is a separate service that exposes a *batchProve* and a *batchVerify* endpoint, mainly used by the L2 node (Section 1.1.2) to prove batches of transfers. Under the hood, the Prover uses a zero-knowledge proving system that computes the account balances resulting from the transfers within a batch and a proof of correct computation.

1.1.4. Data Store

The data store is a persistent storage that stores the performed transfers and the account balances whose state representation is stored on the blockchain. To achieve more failsafe and reliable availability of the data, it is replicated sufficiently often by so-called standbys (replicas). In the case of failure, standbys can be used as new primary stores ([3]).

1.1.5. L1 State/ Verifier Contract

The L1 State and Verifier Contracts are responsible for verifying and performing updates of the Merkle root of account balances. They can be two separate contracts or a single contract with several endpoints. A state update only happens if the updated state was verified successfully beforehand. The contracts are called by the L2 node by creating according transactions and submitting them to an L1 node.

For the Merkle tree root to have an initial value, the State Contract will be initialized with a deposit. This initial deposit then becomes the balance of the system.

1.2. APIs

The following section proposes a possible API of the previously described components.

1.2.1. Client CLI

[1]The state representation is the Merkle root, see Section 2.

Name	Arguments	Return Value	Description
get_balance	accountId: AccountID	balance: UnsignedINT	Returns a user's L2 account balance. The UnsignedINT type should be a type that allows for safe money computations (Read [4])
transfer	sender: AccountID, recipient: AccountID, amount: UnsignedINT, token_id: TokenID, nonce: Nonce, key_pair: KeyPair	transferId: TransferID	Creates, signs and submits an L2 transfer to the L2 node. A cryptographic nonce should be used in order to prevent replay attacks.
getTransfer	transferId: TransferID	transfer: TransferState	Returns the status of a given transfer: Cancelled, ZKP in progress, batch proof in progress, or "posted on L1 with blockhash X"
deposit	depositor: AccountID, amount: UnsignedINT, token_id: TokenID, key_pair: KeyPair	transaction: Transaction	Creates an L1 deposit transaction, by first asking the L2 node to create an according L1 transaction for us, afterward signing it on the client-side and then submitting it to L1 through the L2 node
withdraw	withdrawer: AccountID, amount: UnsignedINT,	transaction: Transaction	Creates an L1 withdraw transaction, by first asking the L2 node

	token_id: TokenID key_pair: KeyPair		to create an according L1 transaction for us, afterward signing it on the client-side and then submitting it to L1 through the L2 node
verify	proof: Proof, public_inputs: PublicInputs	result: VerifyResult	Returns whether a proof is legitimate or not

1.2.2. L2 Node

Name	Arguments	Return Value	Description
get_balance	accountId: AccountID	balance: UnsignedINT	Returns a user's L2 account balance. The UnsignedINT type should be a type that allows for safe money computations (Read [4])
transfer	sender: AccountID, recipient: AccountID, amount: UnsignedINT, token_id: TokenID, nonce: Nonce, signature: Signature, sender_pub_key: PubKey	transferId: TransferID	Schedules an L2 transfer, and returns a transfer-ID
getTransfer	transferId: TransferID	transfer: TransferState	Returns the status of a given transfer: Cancelled, ZKP in progress, batch proof in progress,

			or “posted on L1 with blockhash X”
deposit	depositor: AccountID, amount: UnsignedINT, token_id: TokenID	transaction: Transaction	Creates an L1 deposit transaction containing an according Merkle tree root update and accompanying metadata needed to verify the update based on the depositor’s account ID and the amount. The returned transaction is an L1 transaction that has to be signed by the depositor on the client-side.
withdraw	withdrawer: AccountID, amount: UnsignedINT, token_id: TokenID	transaction: Transaction	Creates an L1 withdraw transaction containing an according Merkle tree root update and accompanying metadata needed to verify the update based on the withdrawer’s account ID and the amount. The returned transaction is an L1 transaction that has to be signed by

			the withdrawer on the client-side.
submitTransaction	signedTransaction: SignedTransaction	submitResult: SubmitResult	Forwards a signed L1 transaction and submits it to the L1 for execution.
verify	proof: Proof, public_inputs: PublicInputs	result: VerifyResult	Returns whether a proof is legitimate or not

1.2.3. L1 State/ Verifier Contract

Name	Arguments	Return Value	Description
verify_deposit	depositor: AccountID, amount: UnsignedINT, token_id: TokenID, updated_account_balances: MerkleRootHash, update_metadata: MerkleRootUpdateMetadata, signature: Signature		• Verifies that the transaction contains the same amount of tokens used to update the account balance • Moves the amount from the depositor's L1 account to the account owned by the system • Verifies the depositor's signature • Verifies the new Merkle root given the public inputs and metadata • Updates the system's on-chain state on successful verification
verify_withdraw	withdrawer: AccountID, amount: UnsignedINT, tokenId:		• Verifies that the transaction moves the same amount of

	TokenID, updated_account_balances: MerkleRootHash, update_metadata: MerkleRootUpdateMetadata, signature: Signature		tokens used to update the account balance • Moves the amount from the account owned by the system to the withdrawer's L1 account • Verifies the withdrawer's signature • Verifies the new Merkle root given the public inputs and metadata • Updates the system's on-chain state on successful verification
verify_transfers	proof: Proof, publicInputs: PublicInputs		• Verifies that the proof computed from performing batch transfers on the L2 is valid • Updates the system's on-chain state on successful verification

1.3. Component Interaction

Figures 2 and 3 show sequence diagrams for processing a user's deposit and transfer requests, respectively.

1.3.1. Deposit Sequence Diagram

Depositing funds to user `Bob`'s account is divided into three phases, which are modelled in the following sequence diagrams.

In the first phase (Figure 2) users submit their deposit requests through the client CLI to the L2 node. The L2 node creates an L1 transaction (*Deploy*) containing the

according Merkle root update and the update metadata, which can be used to verify that the update was done correctly. More precisely, the *Deploy* contains a *Session* that executes the validation of the Merkle tree update, performs the state transition and transfer the funds from the user's L1 account (purse) to the systems L1 account. This *Deploy* needs to be signed by the user before submitting, this is achieved by making use of the L1′s SDK on the client-side.

After submitting, the L1 smart contracts take care of validating the new Merkle root, updating the system's state, and transferring the funds (Figure 3).

Lastly (Figure 4), the L2 node gets notified after the *Deploy* was processed successfully. The node then commits the updated state to the data store. After sufficient time has passed, the user can query its account balance using the client CLI.

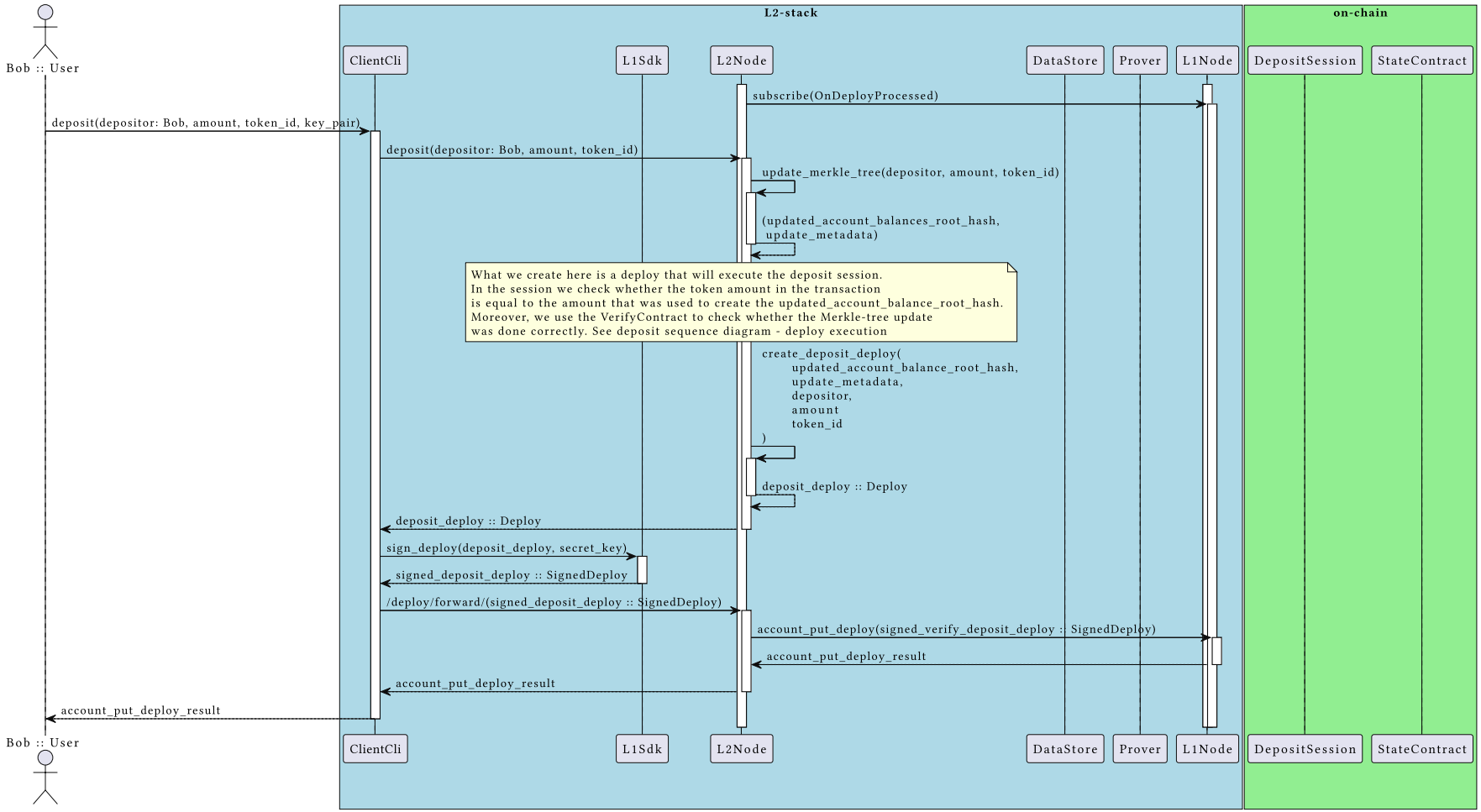

Figure 2: Deposit: User submits a deposit to L2 which gets forwarded to L1.

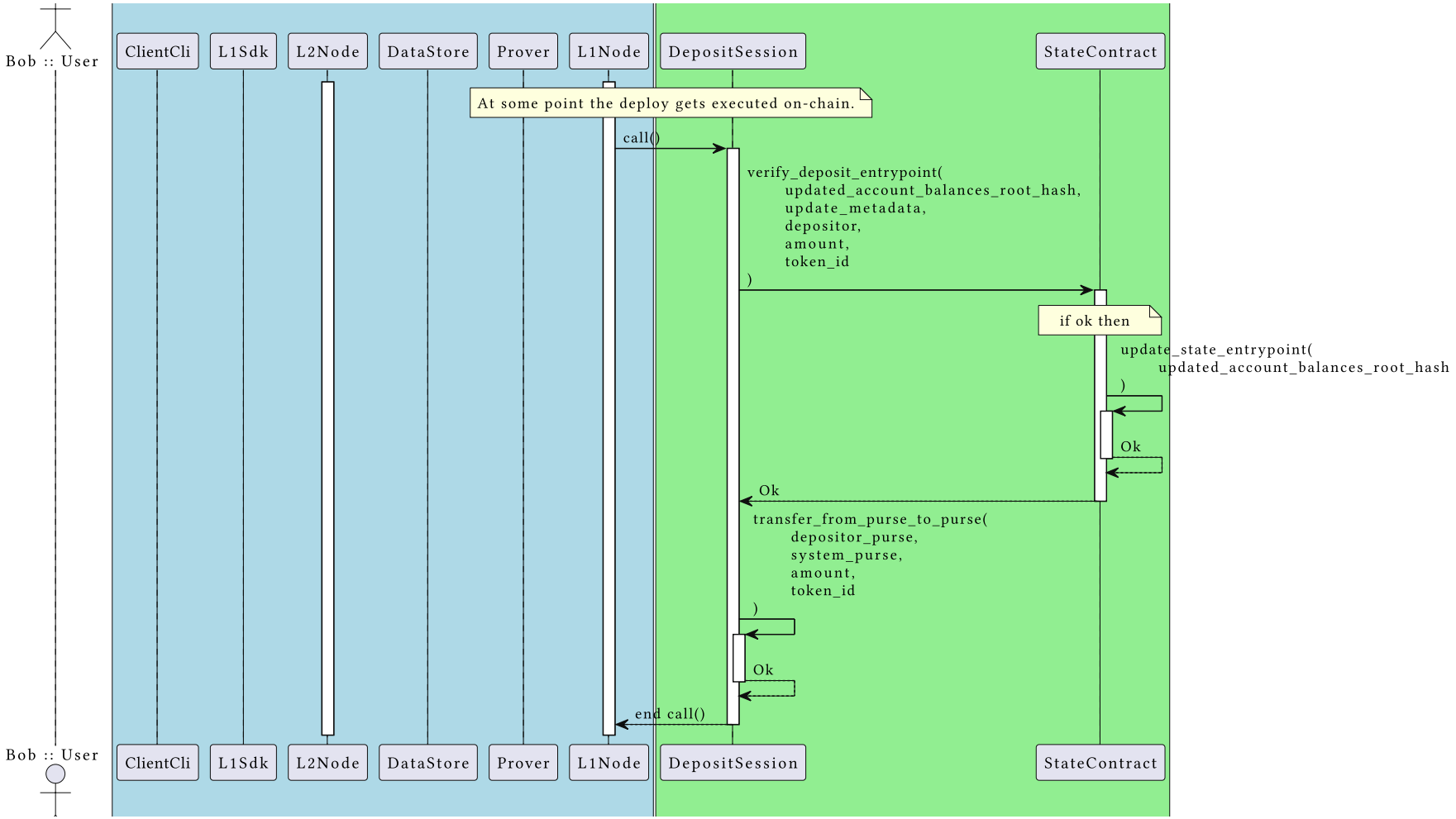

Figure 3: Deposit: Execution of the *Deploy* on L1.

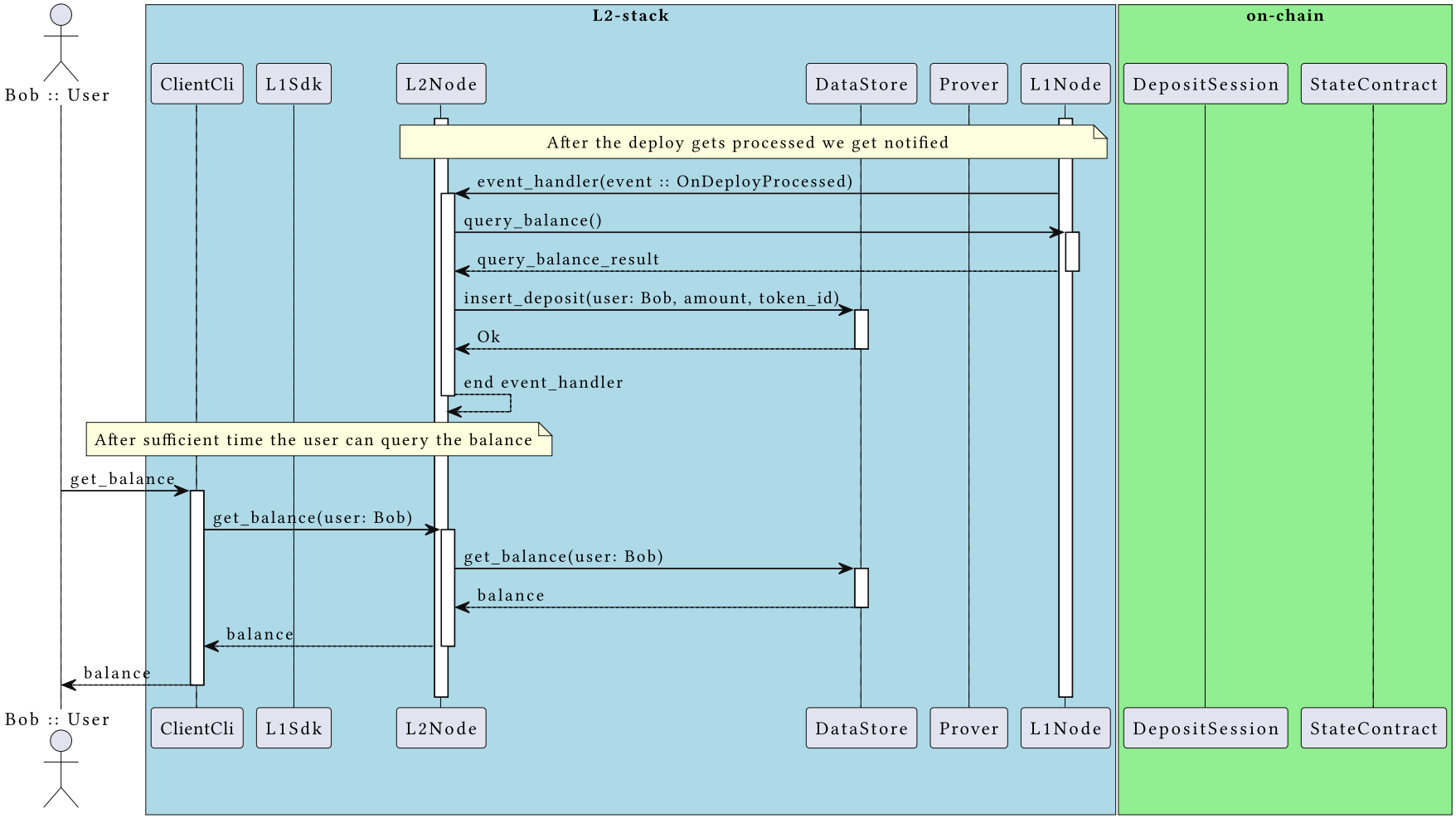

Figure 4: Deposit: Notifying the L2 after succcessfull on-chain execution.

1.3.2. Transaction Sequence Diagram

Transfering funds from user `Bob` to a user `Alice` can be divided into four phases, which are modelled in the following sequence diagrams.

In the first phase (Figure 5) users submit their transfer requests through the client CLI to the L2 node. The L2 node accumulates the transfer requests and checks for independence. In addition, the L2 node will check that the batch proof which is going to be computed next, a valid nonce.

After `t` seconds or `n` transactions (Figure 6), the L2 node batches the transfers, creates a proof of computation, and the according *Deploy* which will execute the validation and the state transition on-chain.

After submitting, the L1 smart contracts take care of first validating the proof and updating the state (Figure 7).

Lastly (Figure 8), the L2 node gets notified when the *Deploy* was executed successfully. The node then commits the updated state to the data store. After sufficient time has passed, the users can query their account balances using the client CLI

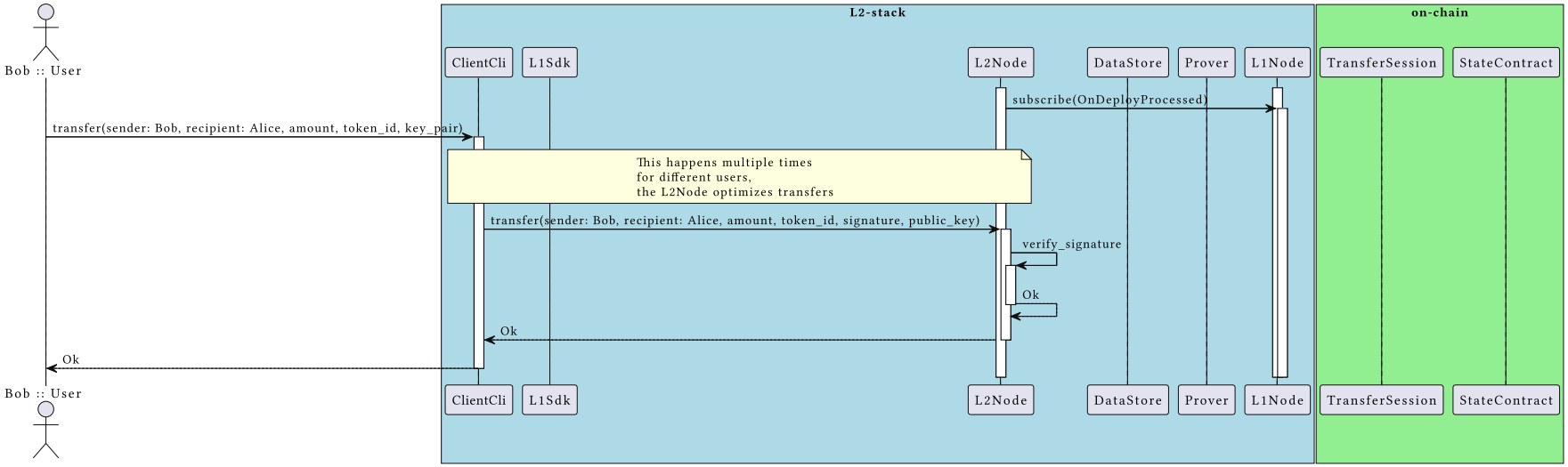

Figure 5: Transfer: User submits a transfer to L2.

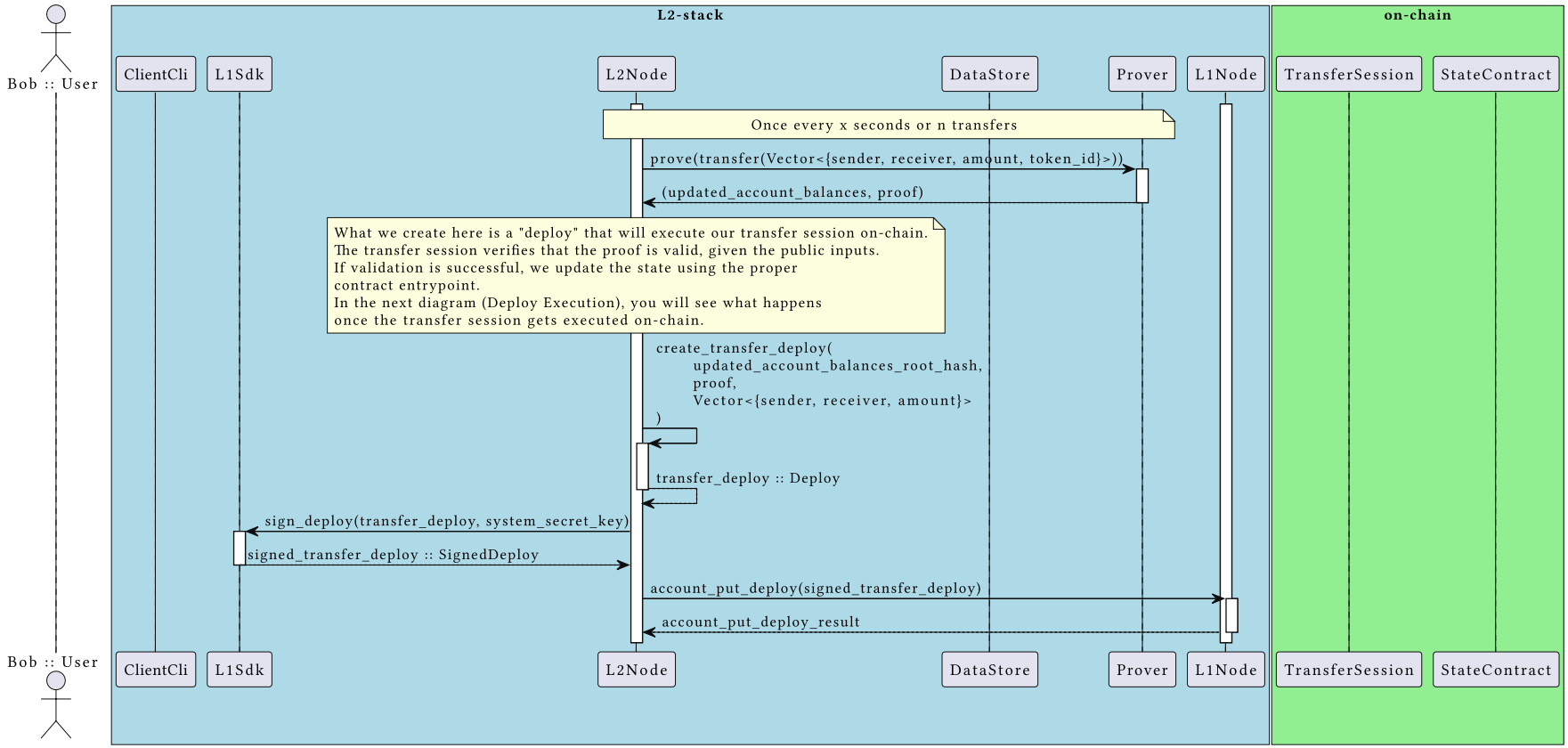

Figure 6: Transfer: Proving and submitting the proof.

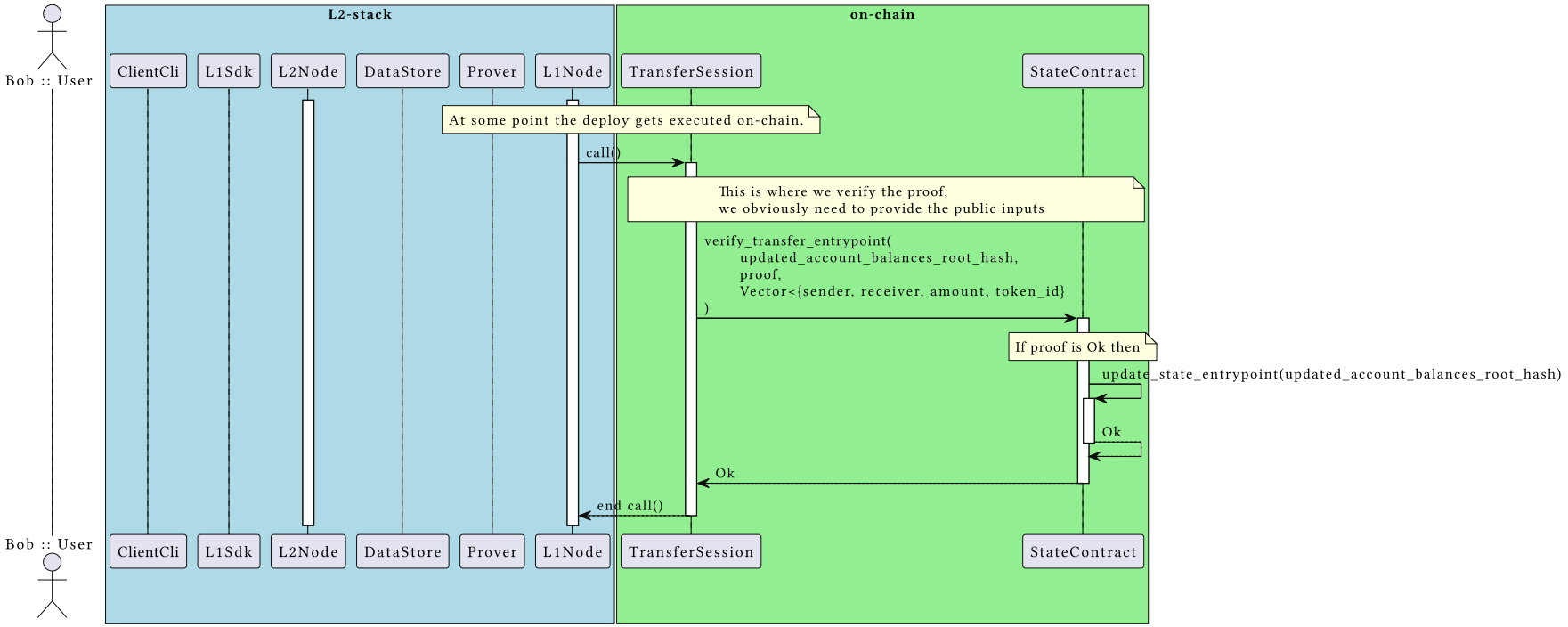

Figure 7: Transfer: Execution of the *Deploy* on L1.

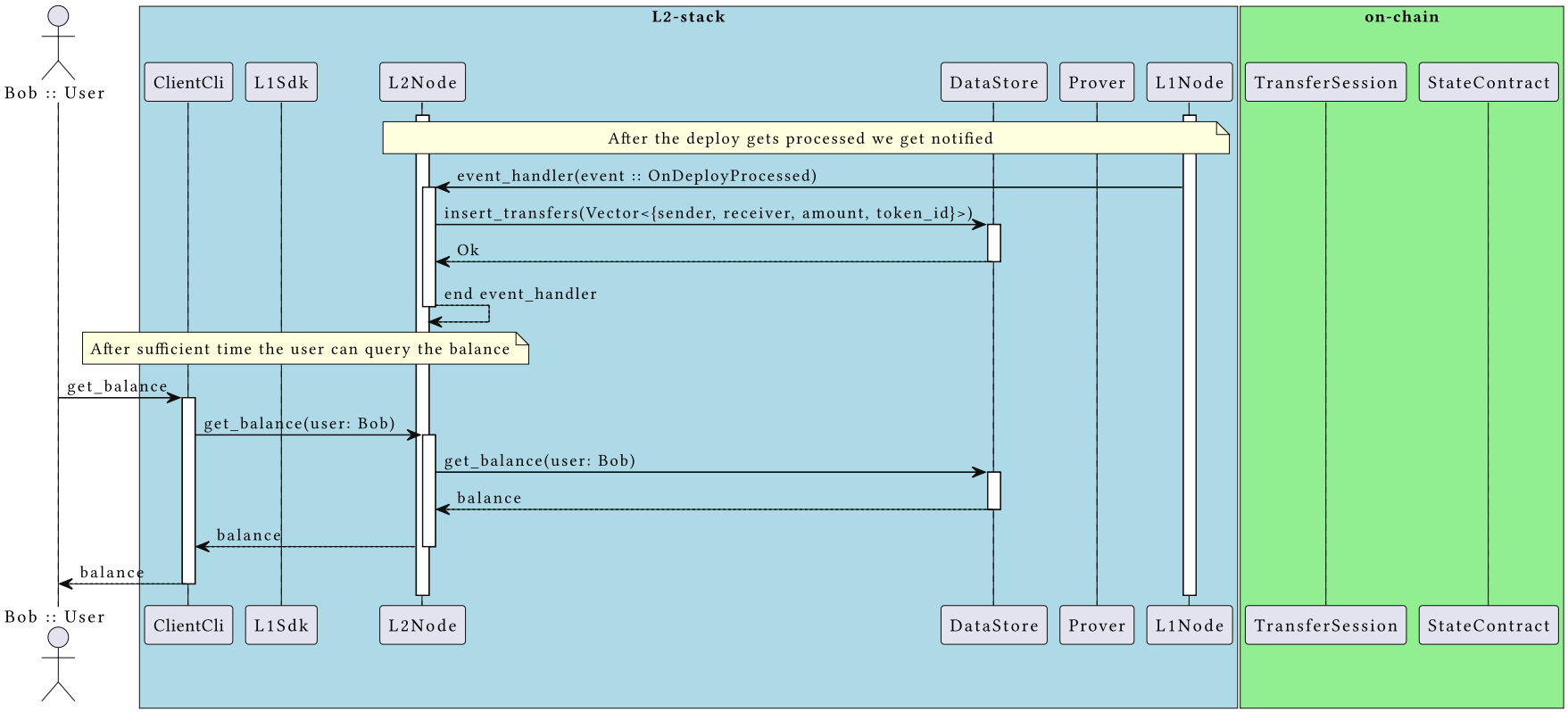

Figure 8: Transfer: Notifying the L2 after succcessfull on-chain execution.

2. Glossary

Validium Please refer to [5] and [6]

L1 The Casper blockchain as it currently runs.

L2 A layer built on top of the Casper blockchain, which leverages Casper's consensus algorithm and existing infrastructure for security purposes while adding scaling and/or privacy benefits

Nonce/ Kairos counter A mechanism that prevents the usage of L2 transactions more than once without the user's permission. It is added to each L2 transaction, which is verified by the batch proof and L1 smart contract. For an in-depth explanation, see [7].

Zero knowledge proof (ZKP) Is a proof generated by person A which proves to person B that A is in possession of certain information X without revealing X itself to B. These ZKPs provide some of the most exciting ways to build L2s with privacy controls and scalability. [8]

Merkle trees Are a cryptographic concept to generate a hash given a dataset. It allows for efficient and secure verification of the contents of large data strutures. [9]

Bibliography

[1] Nick Van den Broeck, "Why build a centralized l2?." [Online]. Available: https://github.com/cspr-rad/kairos-spec/blob/main/blogposts/centralized-l2.md

[2] Nick Van den Broeck, "Merkle trees: How to update without a zero-knowledge proof?." [Online]. Available: https://github.com/cspr-rad/kairos-spec/blob/main/blogposts/merkle-tree-updates.md

[3] Nick Van den Broeck, "Data redundancy." [Online]. Available: https://github.com/cspr-rad/kairos-spec/blob/main/blogposts/data-redundancy.md

[4] Tom Sydney Kerckhove, "How to deal with money in software." [Online]. Available: https://cs-syd.eu/posts/2022-08-22-how-to-deal-with-money-in-software

[5] Ethereum, "Validium." [Online]. Available: https://ethereum.org/en/developers/docs/scaling/validium

[6] Nick Van den Broeck, "ZK validium vs. Rollup." [Online]. Available: https://github.com/cspr-rad/kairos-spec/blob/main/blogposts/validium-vs-rollup.md

[7] Nick Van den Broeck, "L2 transaction uniqueness." [Online]. Available: https://github.com/cspr-rad/kairos-spec/blob/main/blogposts/L2-Tx-uniqueness.md

[8] “Zero knowledge proof.” [Online]. Available: https://en.wikipedia.org/wiki/Zero-knowledge_proof

[9] “Merkle tree.” [Online]. Available: https://en.wikipedia.org/wiki/Merkle_tree