Lien L2 Alpha Specification

Lien Protocol

May 2021

1 About This Document

The Lien L2 Alpha is the world's first Layer-2 option marketplace. As a Layer-2 (L2) protocol, we adopted ZK-Rollup and developed our system with the bellman Community Edition [6], which is one of the most sophisticated library to implement zero-knowledge proof systems.

In our marketplace, a user takes four actions: registration, deposit, trade, and withdrawal. After the registration of the L2 account, the deposit of ETH, and the issuance of SBTs and LBTs in L1, the user can trade the option tokens with almost zero without compromising the security level in L2.

In this document, we explain the detailed specification of the Lien L2 Alpha. The document is written for crypto-engineers who are familiar with the basics of cryptography (such as the zero-knowledge proof).

2 Circuits

Our ZK-Rollup system has two circuits: a **process circuit** and a **transform circuit**. In the beginning, the aggregator generates proofs for the process circuit. On the maturity date, the aggregator suspends the process circuit and begins to generate proofs for the transform circuit. These proofs are verified by a smart contract, called the **verifier contract**.

2.1 Process Circuit

The process circuit validates three types of transactions: register transactions, deposit transactions, and trade transactions. Every time transactions are validated, the user's state in L2 is updated.

These states are committed to a sparse Merkle tree, which we call a state tree. The state tree is constructed using a ZK-friendly (but non-gas-friendly) hash algorithm, MiMC-7. The validity of the state transition and the construction of the state tree are proved with the zero-knowledge proof systems, and therefore no one can steal user's option tokens or perform double-spending. Moreover, anyone can reproduce the valid states in L2 because all transactions are available on L1, which solves the data availability problem [1].

Each process circuit handles 160 transactions. This number is fixed because R1CS cannot express dynamic loops. If there are less than 160 transactions, dummy transactions are padded.

2.2 Transform Circuit

The transform circuit transforms the state tree into the **exit tree**. The exit tree is constructed using a gas-friendly hash algorithm, Keccak-256. Therefore, users can prove their states without paying a high gas fee.

The transform circuit consists of two procedures. First, it verifies the existence of users' states in the last state tree. Second, it constructs a new exit tree from the states. These procedures guarantee that the state tree and the exit tree hold the equivalent states. Each exit tree contains up to 256 users' states to restrict the number of constraints. If more than 256 users are registered, multiple exit trees are constructed.

3 Hash Functions

Hash computations are the most expensive computations that require a large number of constraints in the circuit or large consumption of gas in the smart contract. We use two efficient methods to compute hash functions: SlicedKeccak256 and Chain Hash.

3.1 SlicedKeccak256

SlicedKeccak256 is the hash function that we mainly use in our implementation. It takes the head of 31 bytes from the output of the **Keccak256** hash function. We do not use the whole of the 256-bit output to avoid an overflow problem. Since the order of finite field on the computation in BN256 elliptic curve is smaller than 2^{256}, 256-bit numbers may be outside of the range. By taking the first 31 bytes (i.e., 248 bits) of the hash value, we avoid this problem.

The input data size should be 64 bytes. If the input data size is insufficient, then the input data is padded with zero bits.

3.2 Chain Hash

Throughout this implementation, we often want to compute the hash value of large data, while SlicedKeccak256 can only accept 64-byte data as an input. The **Chain Hash** is an aggregated hash computed from an array of fixed-size bytes data. We typically consider an array of hash values, which we refer to a **hash array**. In this section, we describe how we can compute an aggregate hash of a hash array efficiently.

In the verifier contract, we compute the hash value of the all transaction data (called the all transaction hash, described in section 4.16) using this Chain Hash scheme. The $(n+1)$-th value of Chain Hash is an output of SlicedKeccak256 computed from the n-th value of Chain

Hash and n-th value in the hash array:

$$\begin{aligned} ChainHash_0 &= 0 \\ ChainHash_{n+1} &= Hash(ChainHash_n || HashArray[n]) \end{aligned}$$

The first value of Chain Hash ($ChainHash_0$) is defined as zeros whose length is 31 bytes. Since $HashArray[n]$ has fixed-size bytes (as it is a hash value), the input of the right-hand side has a fixed size. Accordingly, it is a valid input of our hash function, SlicedKeccak256. When there are n non-dummy transactions ($n \leq 160$), the $ChainHash_n$ is adopted as the all transaction hash.

In the circuit, the same method is not applied since R1CS cannot express dynamic loops. Therefore, the $ChainHash_{160}$ must always be adopted regardless of the number of non-dummy transactions. To hold the consistency of the Chain Hash, we ignore the Chain Hash computed with the hash of the dummy transaction ($DummyHash$):

$$\begin{aligned} ChainHash_0 &= 0 \\ IsDummyFlag_n &= (HashArray[n] == DummyHash) \\ ChainHash_{n+1} &= Hash(ChainHash_n || HashArray[n]) * (1 - IsDummyFlag_n) \\ &+ ChainHash_n * IsDummyFlag_n \end{aligned}$$

It is obvious that the previous Chain Hash is inherited to the new Chain Hash in the case that $HashArray[n] == DummyHash$; therefore the verifier contract can compute the $ChainHash_n$ equivalent to the $ChainHash_{160}$ in the circuit, with ignoring dummy transactions. (Figure 1)

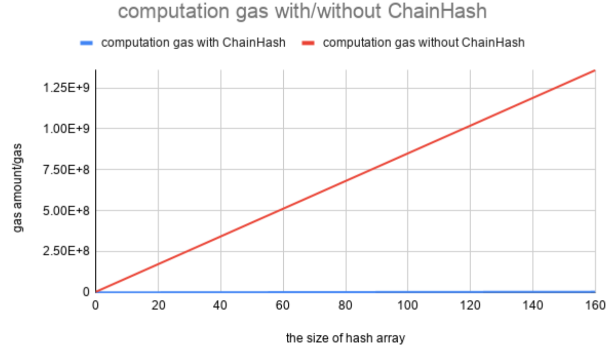

Figure 1: computation gas with/without ChainHash.

4 Data Type Definitions

In this section we explain the definitions of data types that appeared in our implementation. Each data has a bits-data form and can be encoded into bits data.

4.1 UId

UId is a user's id in L2 (**UId**) generated in the registration process. It is a non-negative integer and issued in sequential order. The bits-data form of UId is simply 16 bits data:

Table 1: The bits-data form of UId

UId
16 bits

4.2 L2 PublicKey

A user's public key in L2 (**L2 PublicKey**) is a point in Baby Jubjub Elliptic Curve [8]. It consists of the x-coordinate (254 bits) and the corresponding y-coordinate (254 bits). Since the sum of the two bits size is not a multiple of eight, zero bits are inserted into the bits-data form of L2 PublicKey:

Table 2: The bits-data form of L2 PublicKey

x-coordinate	y-coordinate	zero bits
254 bits	254 bits	2 bits

4.3 L2 Address

A user's address in L2 (**L2 Address**) is computed by taking the head of 20 bytes from the output of SlicedKeccak256 hash. function. The taken data itself is the bits-data form of L2 Address:

Table 3: The bits-data form of L2 Address

L2 address
160 bits

Note that its size and bits-data form is the same as those of the user's Ethereum address in L1 (**L1 Address**).

4.4 Nonce

A user's **nonce** denotes the number of the user's transactions issued in L2. It is a non-negative integer and begins from zero. The bits-data form of nonce is simply 16 bits data:

Table 4: The bits-data form of nonce

Nonce
16 bits

4.5 Balance

A user's balance denotes the balance of his option token in L2. Since there are two types of tokens, namely SBT and LBT, the user has the **SBT balance** and the **LBT balance**. Each bits-data form is 32 bits data:

Table 5: The bits-data form of balance

Balance
32 bits

4.6 L2 State

A user's **state** in L2 consists of UId, L1 Address, L2 Address, nonce, SBT balance, and LBT balance. They are updated by the corresponding transactions in L2. Its bits-data form is a combination of each bits data:

Table 6: The bits-data form of state

UId	L1 Address	L2 Address	nonce	SBT balance	LBT balance
16 bits	160 bits	160 bits	16 bits	32 bits	32 bits

4.7 State Hash

A state hash is an output of SlicedKeccak256 computed from the bits data of the state. Since the size of the state is 416 bits but the required input size of SlicedKeccak256 is 512 bits, 96 zero bits are padded into the state:

Table 7: The input data to compute the state hash

UId	L1 Address	L2 Address	nonce	SBT balance	LBT balance	zero bits
16 bits	160 bits	160 bits	16 bits	32 bits	32 bits	96 bits

4.8 State Tree

A **state tree** is a sparse Merkle tree [7] that contains users' states. Its key is specified with the corresponding UId and its state hash is committed as its value. The root of the state tree (**state root**) is stored on the L1 storage, and updated by L2 transactions. (Note that we implemented the sparse Merkle tree in Rust based on the iden3 implementation [5].)

4.9 Price Data

A **price data** is used to report the price of LBT/SBT (LBT price in SBT) to L2 as a price oracle. The L1 contract obtains the ETH price from ChainLink[1] and calculates the theoretical price of LBT/SBT by solving the Black–Scholes formula. Therefore, the price data specifies the round id of Chainlink and the corresponding calculated price to guarantee that the price used in L2 is not altered by the aggregator.

The round id and the price are non-negative integers and the size of each data is 32 bits. If both data is zero, we call it **dummy price data**.

The bits-data form of the price data is a combination of the round id and the price:

Table 8: The bits-data form of the price data

RoundId	Price
32 bits	32 bits

4.10 All Price Hash

An all price hash is an output of Chain Hash computed from the array of the hash of the price data, i.e.

$$AllPriceHash = ChainHash(SlicedKeccak256(Bytes(PriceData))[]).$$

4.11 Register Data

A **register data** is used to report the registration event to L2. It has the registered user's UId, L1 Address, and L2 Address. Its bits-data form is a combination of these data:

Table 9: The bits-data form of the register data

UId	L1 Address	L2 Address
16 bits	160 bits	160 bits

[1] https://chain.link/.

The output of SlicedKeccak256 computed from the bits data of the register data is specially called **a register hash** in our implementation, i.e.

$$RegisterHash = SlicedKeccak256(Bytes(RegisterData)).$$

It is contained in other data types such as deposit data in section 4.13.

4.12 All Register Hash

An all register hash is an output of Chain Hash computed from the array of the register hash, i.e.,

$$AllRegisterHash = ChainHash(RegisterHash[]).$$

4.13 Deposit Data

A **deposit data** is used to report the deposit event to L2. It contains the user's register hash and the quantity of deposited ETH. In the process circuit, the deposit data is reconstructed based on the information of the deposit transaction, that is, the UId, the L1 Address, the L2 Address, and the quantity to increase the SBT/LBT balances. If a malicious user specified a larger quantity in the deposit transaction than the deposited ETH quantity, the reconstructed deposit data is not equivalent to that in L1 and such inconsistency is detected by a mechanism of Entry Hash described in section 4.17.

The bits-data form of the deposit data is a combination of the register hash and the amount of the deposited ETH:

Table 10: The bits-data form of the deposit data

Register Hash	Quantity
248 bits	32 bits

4.14 All Deposit Hash

An all deposit hash is an output of Chain Hash computed from the array of the hash of the deposit data, i.e.

$$AllDepositHash = ChainHash(SlicedKeccak256(Bytes(DepositData))[]).$$

4.15 L2 Transactions

All of the L2 transactions consist of a **tx type**, an **available value**, and a **non-available value**. The tx type specifies the type of transactions. The available value is a part of the transaction data that is necessary to reproduce the state from the data written on L1 (this property is essential for solving the data availability problem). The non-available value is the rest of the data.

A tx type denotes the type of transaction with 2-bits flags as below:

Table 11: Tx Type

Tx Type Value	Transaction Type
00	SBT Trade Tx
01	LBT Trade Tx
10	Deposit Tx
11	Register Tx

An available value and a non-available value are both used as private inputs in the process circuit, and their contents and size depend on the type of the transaction. The former is, however, only published onto L1 to guarantee data availability.

The **transaction hash** is computed only from the tx type and the available value. Specifically,

$$TransactionHash = SlicedKeccak256(TxType||Bytes(AvailableValue)).$$

Accordingly, the transaction hash can be computed not only in the circuit but also in the

verifier contract.

4.15.1 Trade Transaction

An **SBT transaction** is an L2 transaction to sell SBT. Likewise, an **LBT transaction** is an L2 transaction to sell LBT. The available value of an SBT transaction has the user's UId, the quantity to sell SBT, and the round id corresponding to the specified price data. Anyone can reproduce the new state altered after the transaction only from the published data. Its bits-data form is a combination of the UId, the quantity, and the round id:

Table 12: The bits-data form of the available value in the trade transaction

UId	Quantity	Round Id
16 bits	32 bits	32 bits

Its non-available value has the user's nonce and the digital signature that utilizes EdDSA for the Baby Jubjub Elliptic Curve [2]. Not only the signature but also the nonce is required to prevent replay attacks: a malicious aggregator can submit and execute the same signed transaction in L2 many times without permission if the nonce is not necessary. The non-available value does not have to be published onto L1 as long as their validity is proved with the zero-knowledge proof systems. Its bits-data form is a combination of the nonce and the signature:

Table 13: The bits-data form of the non-available value in the trade transaction

Nonce	Signature
16 bits	1280 bits

4.15.2 Deposit Transaction

A **deposit transaction** is L2 transaction to split the deposited ETH into SBTs and LBTs and increase his SBT/LBT balances. Its available value consists of the user's register hash and the quantity of the deposited ETH. The register hash should be published because the

L1 contract has to verify that the user of the register hash is the same person who deposited the corresponding ETH in L1. Specifically, from the available value the deposit data is reconstructed in the circuit, and invalid data not equivalent to that of L1 is detected. Its bits-data form is a combination of the register hash and the quantity:

Table 14: The bits-data form of the available value in the deposit transaction

Register Hash	Quantity
248 bits	32 bits

Its non-available value is only the user's register data. This is necessary to verify the UId, L1 Address, and L2 Address in the old state in the circuit. Its bits-data form is the same as that of the register data.

Table 15: The bits-data form of the non-available value in the deposit transaction

Register Data
336 bits

4.15.3 Register Transaction

A **register transaction** is L2 transaction to register a new account in L2. Its available value is only the user's register hash. As well as the deposit transaction, it is required to verify that the user of the register hash is the same person who obtained the corresponding UId in L1. Its bits-data form is the bits data of the register hash:

Table 16: The bits-data form of the available value in the register transaction

Register Hash
248 bits

Its non-available value is only the user's register data. This is necessary for the same reason described in the above section. Its bits-data form is also the bits data of the register data:

Table 17: The bits-data form of the non-available value in the register transaction

Register Data
336 bits

4.16 All Transaction Hash

An **all transaction hash** is an output of Chain Hash computed from the array of the transaction hash, i.e.,

$$AllTransactionHash = ChainHash(TransactionHash[]).$$

.

4.17 Entry Hash

The **entry hash** [3] is an output of SlicedKeccak256 to fix the gas amount caused by public input. The use of entry hash is necessary because operations on BN256 elliptic curve is expensive. For example, each scalar multiplication on BN256 elliptic curve needs 6,000 gas [4]. By fixing the length of the input, we can save the gas amount incurred for such computation (Figure 2).

In our implementation, Entry Hash is defined in the following way.

$$EntryHash = Hash\left(\begin{array}{c} AllTransactionHash \,||\, OldStateRoot \,||\, NewStateRoot \,|| \\ AllRegisterHash \,||\, AllDepositHash \,||\, AllPriceHash \end{array}\right).$$

The entry hash is used both on the verifier contract and the circuit. Although the transactions, state roots, register data, deposit data, and price data are published onto L1, the entry hash is only utilized as the public input and the others are private input. If any published-but-private input in the circuit is different from the published data, the entry hash should also become different. Therefore, the entry hash can verify that the published data on L1 is

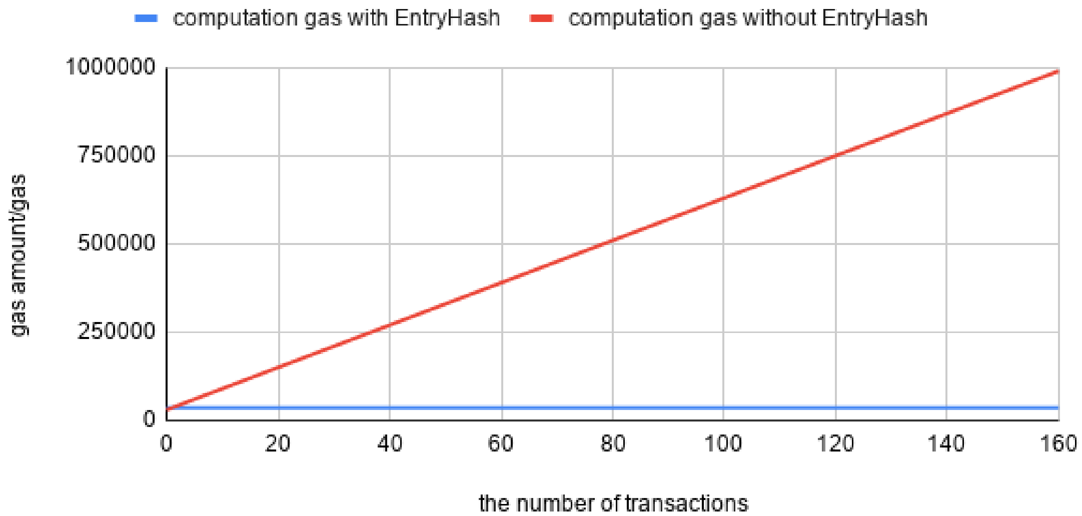

Figure 2: computation gas for the scalar multiplication on BN256 elliptic curve with without EntryHash.

equivalent to the private input in the circuit.

4.18 Overwrite Tree

In our specification, each exit tree contains up to 256 transactions. The number of states in each tree is fixed because the transform circuit cannot process a dynamic loop. This technical limitation causes the following two problems.

1. The gas amount to store the roots of the exit trees in the L1 storage increases linearly with the number of users.

2. The users whose states are committed to the late exit trees cannot withdraw their assets immediately after maturity.

To resolve this problem, we combine all the exit trees and construct one large Merkle tree, called an **overwrite tree**. The new root of the overwrite Tree is computed from its

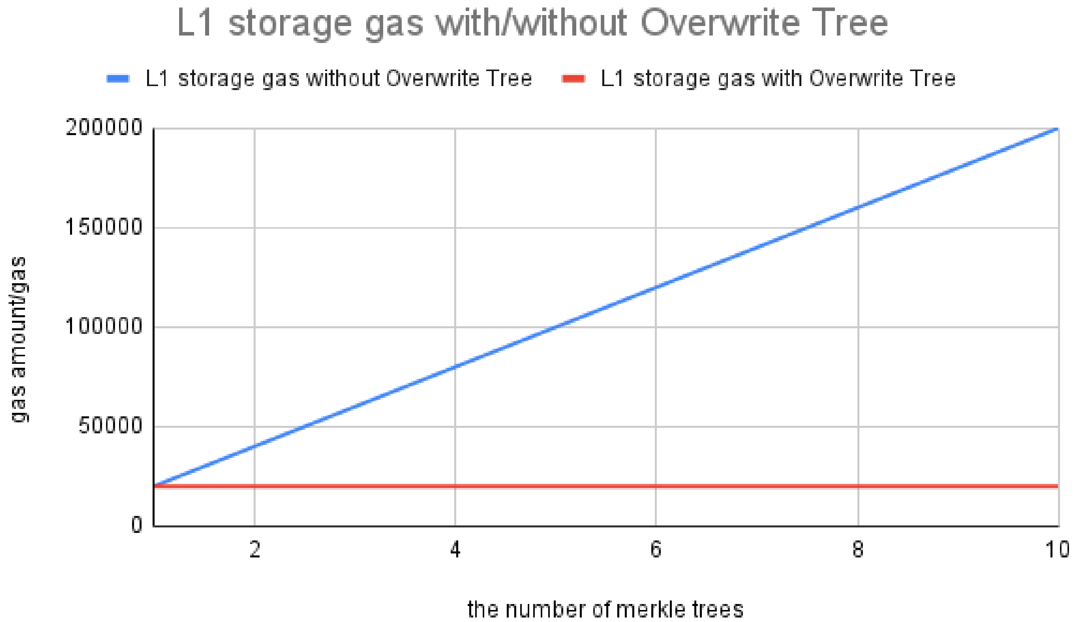

Figure 3: L1 storage gas with/without Overwrite Tree.

previous root and the root of the added Merkle tree:

$$OverwriteTreeRoot_{n+1} = Hash(OverwriteTreeRoot_n || AddedTreeRoot)$$

To prove the existence in the Merkle tree combined into the overwrite tree, two proofs are required.

1. The proof that the last root of the Overwrite Tree is calculable from the root of the specified Merkle tree.

2. The Merkle proof for the specified Merkle tree.

When n Merkle trees are combined into the overwrite tree, the root of i-th Merkle tree is proved with the i-th root of the overwrite tree and the roots of $i + 1 \ldots n$ Merkle trees.

In our implementation, multiple exit trees are combined into the overwrite tree, and its root is only stored in the L1 storage. In this way, the first inefficiency of the transformed circuit is solved (Figure 3).

In addition, the gas amount to verify the first proof increases with the number of added exit trees. Therefore, after all of the exit trees are added, the users whose states are committed to the late exit trees can withdraw their assets with less gas amount. Accordingly, those who committed to late trees can withdraw their profit with lower gas costs. While the second inefficiency of the transformed circuit is not completely resolved, the users in the late exit trees are compensated with the gas costs.

References

[1] Mustafa Al-Bassam, Alberto Sonnino, and Vitalik Buterin. Fraud and data availability proofs: Maximising light client security and scaling blockchains with dishonest majorities, 2018.

[2] Jordi Baylina and Marta Bellés. Eddsa for baby jubjub elliptic curve with mimc-7 hash. https://iden3-docs.readthedocs.io/en/latest/_downloads/a04267077fb3fdbf2b608e014706e004/Ed-DSA.pdf.

[3] Vitalik Buterin. On-chain scaling to potentially ∼500 tx/sec through mass tx validation, 2018. https://ethresear.ch/t/on-chain-scaling-to-potentially-500-tx-sec-through-mass-tx-validation/3477.

[4] Antonio Salazar Cardozo and Zachary Williamson. Eip-1108: Reduce alt bn128 precompile gas costs, 2018. https://eips.ethereum.org/EIPS/eip-1108.

[5] iden3. iden3 implementation of sparse merkle trees, n.d. https://github.com/iden3/circomlib/tree/master/circuits/smt.

[6] matter labs. bellman "community edition", n.d. https://github.com/matter-labs/bellman.

[7] Dahlberg Rasmus, Pulls1 Tobias, and Peeters Roel. Efficient sparse merkle trees, 2016.

[8] Barry WhiteHat, Jordi Baylina, and Marta Bellés. Baby jubjub elliptic curve, 2019. https://iden3-docs.readthedocs.io/en/latest/_downloads/33717d75ab84e11313cc0d8a090b636f/Baby-Jubjub.pdf.