The zkEVM Architecture

Concepts

Polygon zkEVM & Universitat Politècnica de Catalunya (UPC)

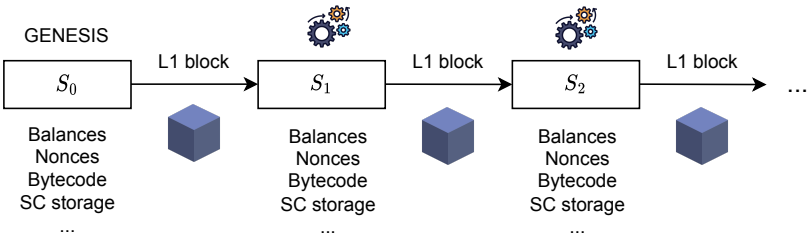
Marc Guzman-Albiol <marc.guzman.albiol@upc.edu>
Jose Luis Muñoz-Tapia <jose.luis.munoz@upc.edu>

Version 1.1

December 12, 2023

Outline

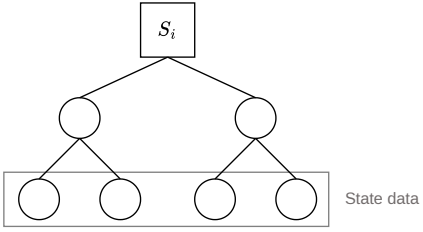
Ethereum Layer 1



Transactions in L1 blocks are **available** and **executed**.

Representation of Each State

- S_i denotes a cryptographic summary of the data of state *i*.
- S_i is implemented as the **root** of a Merkle Tree that includes data items as leaves of the *i*-th state.



Outline

Scaling Blockchain and the Scaling Trilemma

Scaling Blockchain

When we talk about scaling blockchain, we talk about **increasing the number of processed transactions per second**.

- The term **scalability trilemma** was first coined by *Vitalik Buterin* to describe the inherent tension between three properties that a high-performing blockchain platform must have:
  1. Decentralization.
  2. Security.
  3. Scalability.
- The trilemma refers to the belief that blockchain platforms can only achieve two of these three goals effectively.

How to scale?

Approach #1

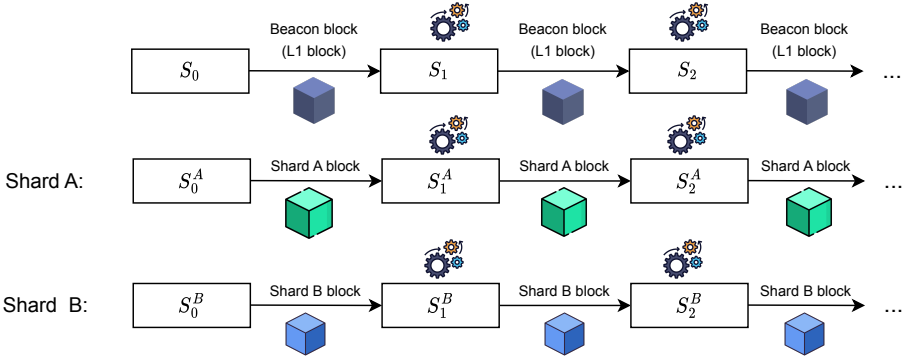
- Increase transactions per block.
- This may lead many blockchain nodes to exhaust their resources.
- Therefore, this may trigger centralization (a network with only powerful nodes).

Approach #2

- Do **sharding**, which means split the burden in shards.
- A node only deals with a *portion* of the burden, i.e. with the operations in its shard.
- There are several approaches within the sharding strategy.

Approach #2.a: Sharding for Data Availability and Execution

- The first approach consists in using the current L1 chain as a *consolidation chain*, which will be renamed as the L1 **beacon chain**.
- Then, use **availability** and **execution** in each shard:



Problems of Sharding with Data_Availability+Execution

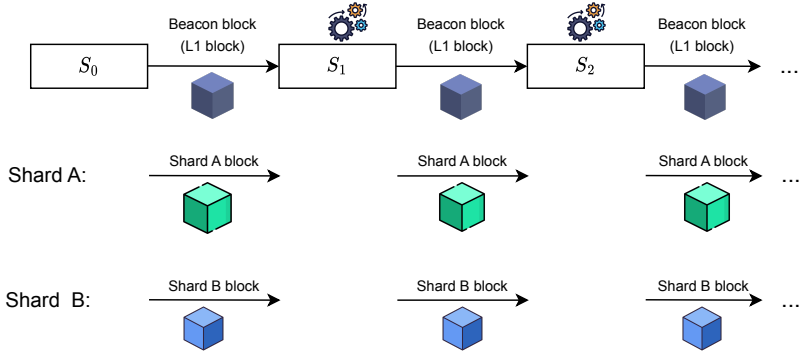
The previous approach is a L1 design with **availability and execution** in each shard.

However, this produces a huge instability in Ethereum L1 specifications:

- L1 Ethereum is responsible for the managements of each of shard's states.
- This includes the **execution** and **inter-shard messaging** when necessary.
- This does not allow L1 **ossification**.

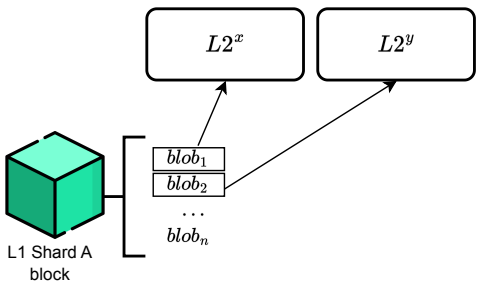
Approach #2.b: Data Availability Sharding and a Single Execution Layer

- In this approach Ethereum specifications provide:
  1. Just **ONE L1 execution layer** giving execution and availability (that is, the current L1).
  2. Data availability sharding scheme (EIP-4844: Shard Blob Transactions).



blobs

Instead of transactions, shard blocks contain **blobs** (binary large objects) that designed to be interpreted by a top layer.



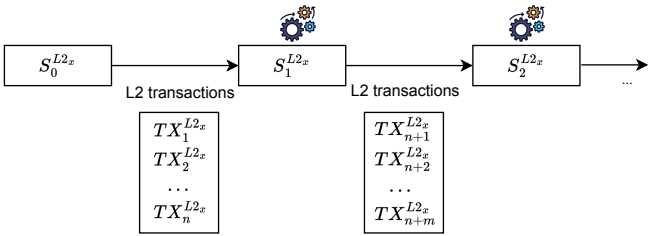
L2 Layers

- New top layers, called **L2 layers**, can be created on top of this L1 machinery.
- L2's define how they manage the state:
  - A payment system with simple transactions.
  - A token transfer system.
  - A system with smart contracts.
- L2's also define how they use L1:
  - The L1 execution layer.
  - The data shards (when available[1]).

[1] Regarding the current situation of data sharding, L1 data shards specification is still under develop (the **EIP 4484** is currently on the "review stage"). For this reason, currently, all the scaling solutions use the availability of the execution layer.

Building a Layer 2

- Let's assume that our layer 2 is called x ($L2_x$).
- The state of the layer x progresses with its L2 transactions.

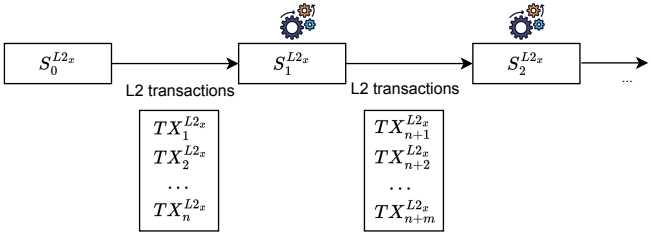


- There are many questions to answer to build an L2:
  a) How users send L2 transactions and who receives them?
  b) How these L2 transactions are made publicly available (if so)?
  c) Who processes the L2 transactions and how, and, when it is publicly considered that a new state is correctly computed?
  d) What type of applications the L2 supports? simple or rich processing?

Outline

Building a Layer 2

- Let's assume that our layer 2 is called x ($L2_X$).
- The state of the layer x progresses with its L2 transactions.

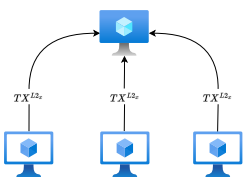


- There are many questions to answer to build an L2:
  a) How users send L2 transactions and who receives them?
  b) How these L2 transactions are made publicly available (if so)?
  c) Who processes the L2 transactions and how, and, when it is publicly considered that a new state is correctly computed?
  d) What type of applications the L2 supports? simple or rich processing?

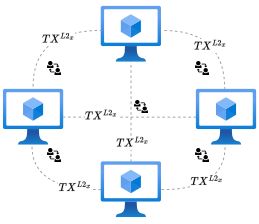
Layer 2 Design: Sending L2 Transactions

Q. How users send L2 transactions and who receives them?

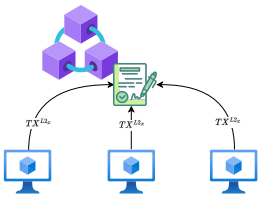
- **Unicast**: an unicast communication with some (centralized) entity.
- **Peer-to-peer**: a peer-to-peer network where everybody (decentralized) receives the L2 transactions.
- **Smart contract**: a smart contract in the L1 execution layer (decentralized).



Unicast



Peer-to-Peer



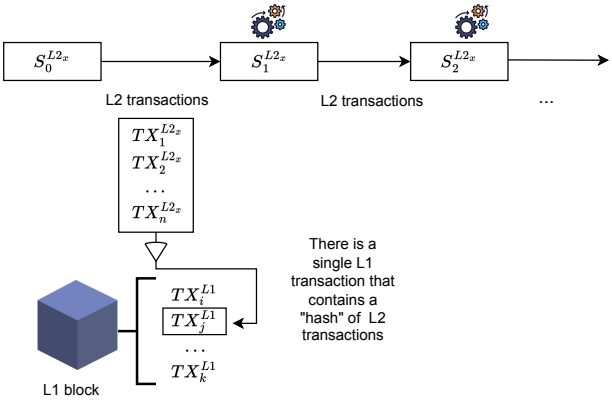
Smart Contract

Layer 2 Design: L2 Data Availability

Q. How are L2 transactions made publicly available (if so)?

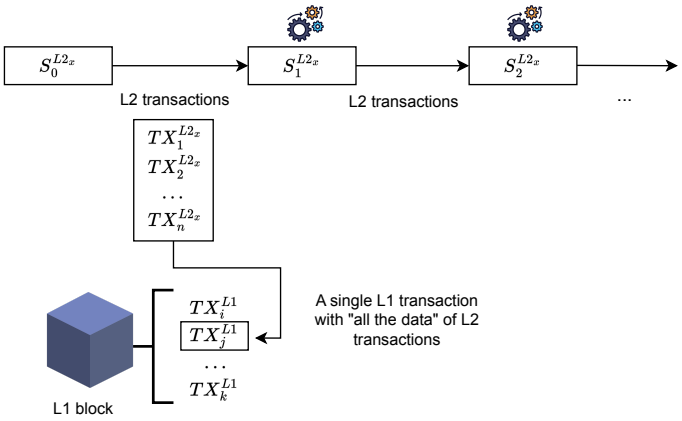
- To achieve L2 data availability in Ethereum, currently, we can proceed in two ways:
  - As a **validium** in which L2 data is managed by a group of **trusted** entities (**data managers**), being this approach far more **cheaper** than writing to L1.
  - As a **rollup**, which writes L2 data in the public L1 Execution layer, meaning that the posted data will be publicly available.
- In the future, with the introduction of the EIP-4844, a third option opens up with **data shards**.

L2 Validiums



In a validium, the L1 transaction only includes a cryptographic summary of the L2 transactions.

L2 Rollups



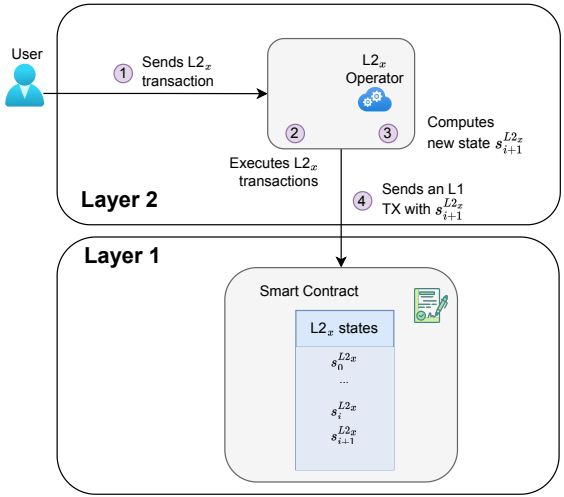
- In this case, to provide data availability, we use a single L1 transaction that contains a batch of L2 transactions.
- This idea is also called a **rollup**, because we ”roll up” a bunch of L2 transactions in a single L1 transaction.

Layer 2 Design: State Computation

Q. Who processes the L2 transactions and how, and, when it is publicly considered that a new state is correctly computed?

- Recall that we need to compute the next L2 state $S_{i+1}^{L2_x}$ from a set of $L2^x$ transactions and the current state $S_i^{L2_x}$.
- We can do that in several ways:
  a) Centralized execution.
  b) Optimistic execution.
  c) Succinct computation verification (zk technology).

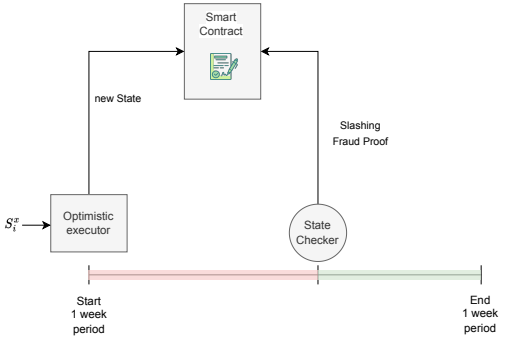
Centralized Execution



- In this approach, the state computation is considered final quickly (takes only ”seconds or minutes”).
- However, this approach has the issue of how “we” (as external entities) dispute the L2 operator about the correctness of an L2 state computation?

Optimistic Execution

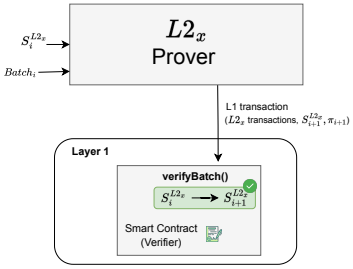
- An **Optimistic L2** provides a decentralized execution mechanism that allows disputing the correctness of the L2 state computation.
- With this approach, there is a (large) period of time to allow anybody to send a **fraud proof**, proving that the state was wrongly computed (for example, a double spending transaction).



- If the Optimistic Executor does its job correctly, it will earn ETH.
- Otherwise, executor is slashed and **state checker** is rewarded (for providing the fraud proof).
- Notice that the progress of the state with optimistic execution is slow, since in general, it takes ”days” to consider the state as final.

Succinct Execution Verification (zk* Systems)

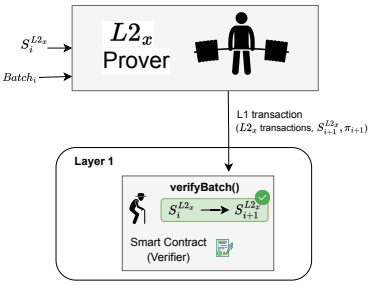
- In the succinct execution verification model, instead of an optimistic executor, we have an execution **prover**.
- The prover can prove that an execution of a set of L2 transaction is correct.
- The set of L2 transactions being proved is called a **batch**.
- The prover uses **Zero-Knowledge (ZK)** technology.



- A smart contract in L1 is the **verifier** of the proof of the batch execution.
- There is no possible dispute since the ZK proof proves that state is correctly computed.
- When the smart contract executes the transaction, we say that the next state, S_{i+1}^{x}, is **consolidated**.
- With this approach, the state computation is quickly considered final since it takes only ”seconds or minutes” to generate and validate the proof.

Remarks about Prover and Verifier

- Regarding the **prover**:
  - It is typically allocated in a cloud service.
  - Uses a considerable amount of resources (mainly RAM and CPU).
- Regarding the **verifier**:
  - The proof size is small (just a few bytes).
  - The verification time is also small (of the order of μs).
  - As a result, a smart contract can indeed verify proofs of L2 batch processing.



Summary of the L2 Scaling Solutions

Scalability solutions for blockchains can be classified by two main dimensions:

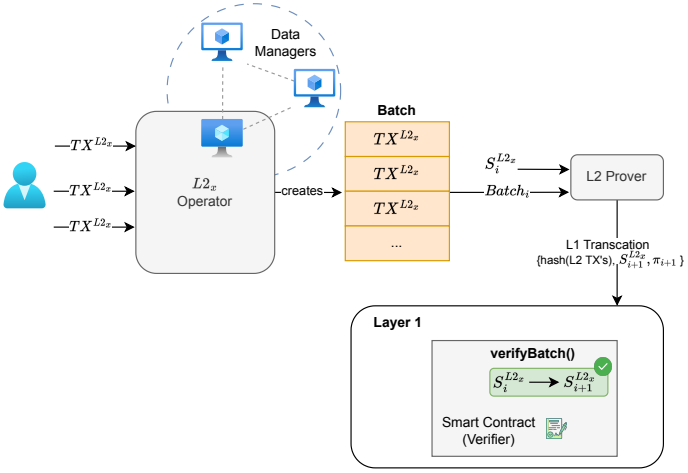
- **Data availability.** Whether the data from the L2 chain is available on-chain (in L1) or off-chain (only in the L2 chain).
- **Mechanism to state correctness.** How the correctness of the L2 chain is guaranteed.

	Validity Proof	Fraud Proof
Data on-chain	zkRollup	Optimistic Rollup
Data off-chain	zkValidium	Plasma

https://l2beat.com/scaling/summary

Remarks about zkValidiums i

- In **zkValidium**, instead of posting all the batch data back to L1, only a cryptographic summary (a *hash*) of it is posted.
- This means that a user cannot retrieve the L2 transactions from L1.
- Instead, the user must ask a *reliable* L2 operator (data manager) for these data.



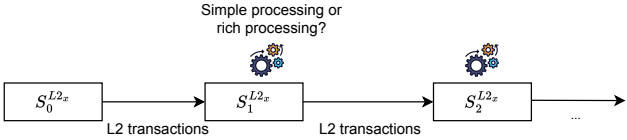
Remarks about zkValidiums ii

In case the operators do not provide the data to the user, still the ZK processing assures that the transactions included in the batch are always correctly processed.

This means that, for example:

- No one can steal your funds.
- You might see an increase in your L2 balance, but you might not know which concrete L2 transaction produced this increase.

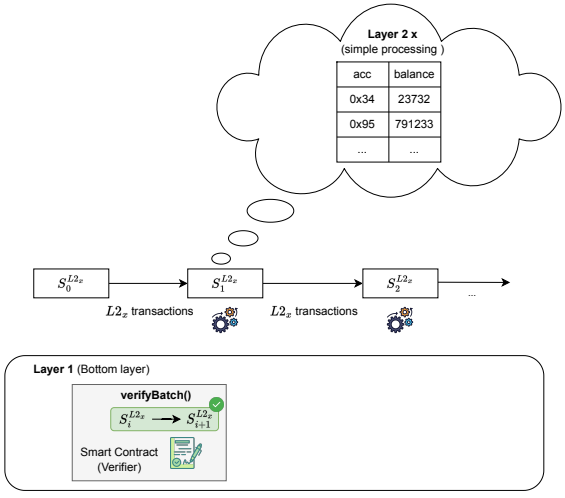
L2 Applications: Simple or Rich Processing



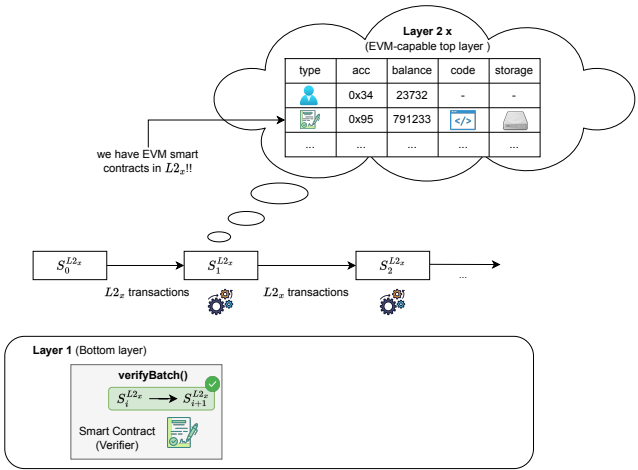
Q. What type of applications the L2 supports?

- **Simple asset transfers**, e.g. tokens or payment network with ”simple processing” of the transactions.
- **General purpose execution virtual machine**, e.g. EVM with ”rich processing” of the transactions.

L2 Simple Processing



L2 Rich Processing (zkEVM)



Are we Scaling with zkRollups?

Q. A last natural question is if we are scaling with rollups based on succinct verification.

The answer is yes, because the smart contract execution resources for verifying a proof are much lower than executing individual transactions within a batch.

In fact, the majority of the cost comes from the data availability, but we can work on improve the costs of data availability.

Addressing data availability costs:

- Succinct verification of **compressed** data (for example, transaction digital signature).
- EIP 4884 **Proto-Danksharing**:
  - Data shards are much cheaper than writing in L1 execution layer.
  - Remark that the L1 execution layer (smart contracts) will have access to blobs in data shards.

Outline

Overview of Transaction Encodings in Ethereum i

- Transactions before EIP155 (**pre-EIP155**):
  - The **raw transaction string** sent to the network is the following:

$$rlp(nonce, gasprice, startgas, to, value, data, r, s, v)$$

  - Where rlp is the run-length-prefix enconding of Ethereum and the values r, s and v are the transaction digital signature.
  - In particular, r and s are 32-byte-values and v is a byte-value.
  - The **to-be-signed-hash** is:

$$to\text{-}be\text{-}signed\text{-}hash = keccak(rlp(nonce, gasprice, startgas, to, value, data))$$

  - The value of v is $\{0, 1\} + 27$, that is, 27 or 28 depending on the parity of the y value of the curve point for which r is the x-value in the **secp256k1** signing process.

Overview of Transaction Encodings in Ethereum ii

- After **EIP155**:
  - The **raw transaction string** sent to the network is the same:

$$rlp(nonce, gasprice, startgas, to, value, data, r, s, v)$$

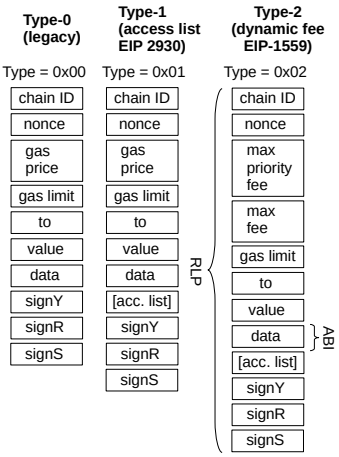
  - However, the **to-be-signed-hash** is the **keccak** hash of the rlp-encoding of 9 elements:

$$to\text{-}be\text{-}signed\text{-}hash = keccak(rlp(nonce, gasprice, startgas, to, value, data, chainid, 0, 0))$$

  - Then, the v of the signature must be set to $\{0, 1\} + CHAIN\_ID * 2 + 35$, where $\{0, 1\}$ is the parity of the y value of the point in the elliptic curve as before.
    E.g. for Ethereum mainnet ($CHAIN\_ID = 1$), the possible values for v are 37 and 38.
    E.g. for zkEVM mainnet ($CHAIN\_ID = 1101$), the possible values for v are 2237 and 2238.
  - The v parameter in EIP155 is a variable length value that can be bigger than one byte.
- The pre-EIP155 signature scheme using $v = 27$ and $v = 28$ remains valid.

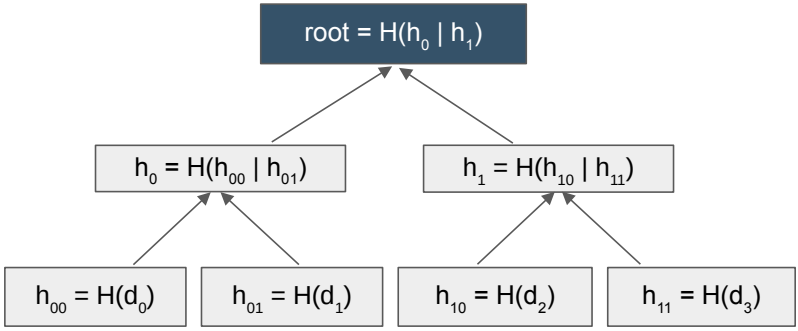
Overview of Transaction Encodings in Ethereum iii

- The **EIP-2718** introduces an envelope for transactions, forcing that all the transactions start with a type.
  - The type for legacy transactions is `0x00`.
  - The type for transactions that include the access list field (**EIP2930**) is `0x01`.
  - The type for transactions with the new dynamic fee scheme (**EIP1559**) is type `0x02`.
- Currently, **zkEVM does not support EIP2718 transactions** (typed transactions), so if we receive such transactions we discard them.



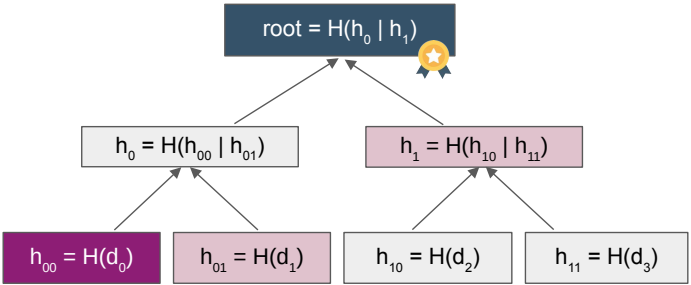
Outline

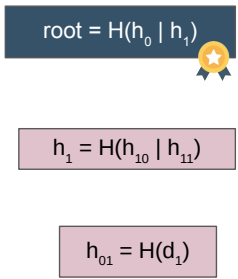
Merkle Tree Concept



- This is a binary tree.
- The root contains contribution from all the leaves and if a single bit is changed in a leaf the root will change.
- For n leaves needs $log_2(n)$ levels.

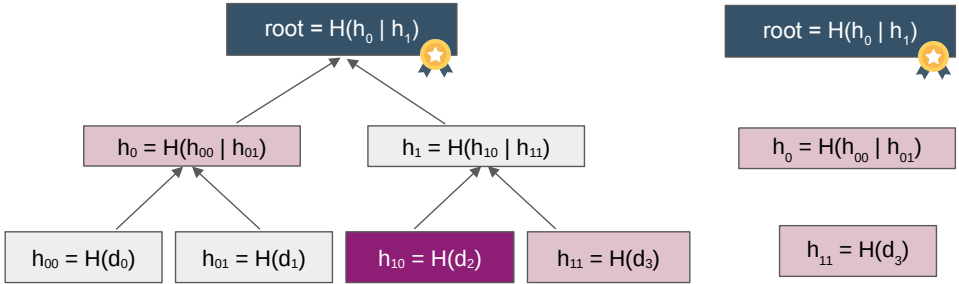
Merkle Proofs i





Proofs are of size $O(log_2(n))$.

Merkle Proofs ii

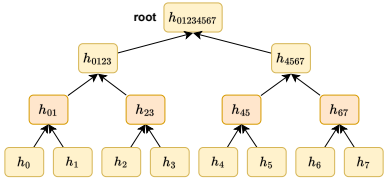


To forge a proof, an attacker needs to find a collision of the hash function, i.e. a different leaf or intermediate node that has the same hash as the one in our tree to have the same root.

Second Pre-image Attack i

If the Merkle hash root is not linked with the tree depth, this enables a second-preimage attack:

- That is, an attacker can create a version of the tree other than the one with the original elements that has the same Merkle hash root.
- In this example, the attacker pretends that the contained elements in the tree are h_{01}, h_{23}, h_{45} and h_{67}.



Second Pre-image Attack ii

- An important note is that this attack can occur even if the underlying hash function has no known security weaknesses: it is inherently a problem in how many Merkle Tree's are constructed.
- Some simple fixes
  a) When computing leaf node hashes, a `0x00` byte is prepended to the hash data, while `0x01` is prepended when computing internal node hashes.
  b) Add the length of the tree in the leaves as part of the data being hashed.
- These attacks often not considered a big concern because whilst the attacker can find input *Y* which generates the same hash value as input *X*, the attacker has no control over the value of *Y* or even its format.
- This means that the input will likely not be interpreted in a useful manner.

Merkle Trees Classification

Merkle Tree

A Merkle Tree is a tree in which every leaf is labelled with the cryptographic hash of a data block, and every node that is not a leaf is labelled with the cryptographic hash of the labels of its child nodes.

We can classify Merkle tree according to four aspects:

a) Tree shape.

b) Data structure.

c) Data inclusion proofs.

d) Tree updatability.

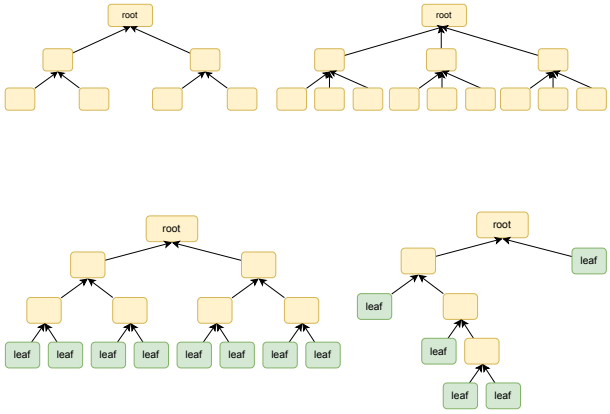
Merkle Trees Classification: Tree Shape

Tree's arity:

- Refers to the number of children that every node that is not a leaf can have.
- Typical examples include:
  - Binary trees (smaller proofs).
  - Ternary trees.
  - Hexadecimal trees.
  - 2-3 trees, and more.

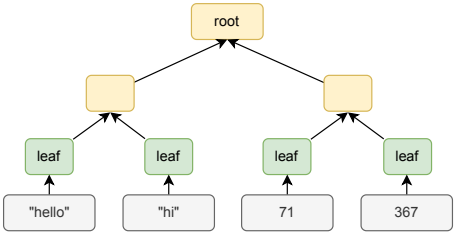
Balanced or unbalanced:

- A tree is balanced if the height of the tree is $O(Log_k n)$.
- Where n is the number of nodes and k is the tree's arity.

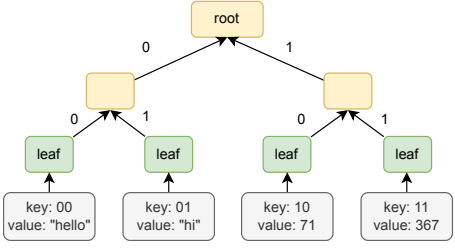


Merkle Trees Classification: Data Structure

- **Unstructured:**
  - Data are not structured according to any specific criteria.

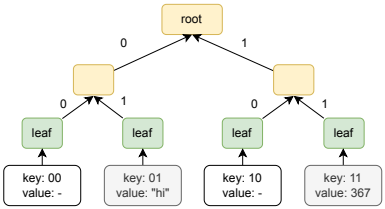


- **Key-value structure:**
  - Data is structured as key-value pairs.
  - The key is used to find the leaf that represents the data.



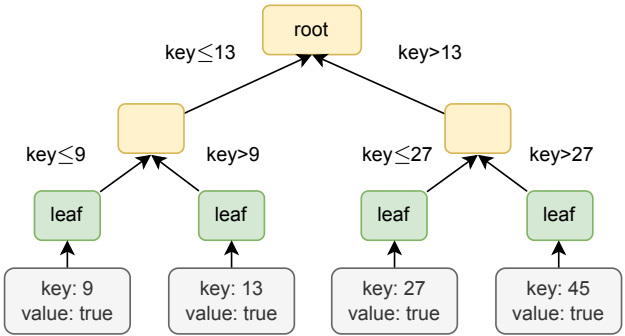
Merkle Trees Classification: Data Inclusion

- **Data-inclusion-only:**
  - The tree that only allows to generate proofs of data inclusion.
  - Unstructured trees only allow this type of proofs.



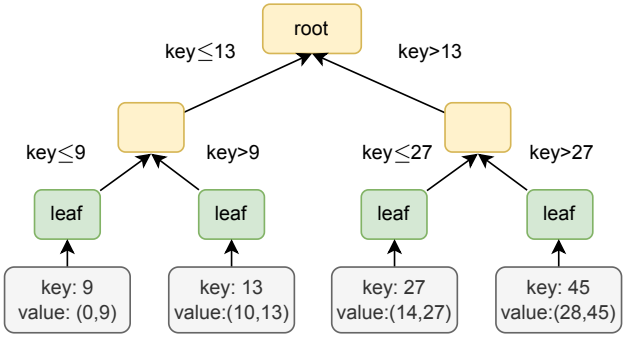
- **Complete:**
  - A tree that allows for non-inclusion proofs that prove that there is no value associated with a given key.
  - These trees are based on the idea of a **complete tree**, i.e. a tree that stores all the possible key-value bindings.
  - In this context, appears the concept of **Sparse Merkle Tree (SMT)**, which is a complete tree with a big (or huge) key space, so it is sparsely populated.
  - The naïve representation of the SMT can have an intractable size, e.g. 2^{256}, and, techniques to compress the required space are applied to efficiently store the tree.
  - In addition, in SMTs, there is typically a default value that is used when the given key has no value associated.

Complete Trees: Example with Ranges i



- The Merkle tree contains four key-values: (9-true,13-true,37-true,45-true).
- We can provide a proof about inclusion but how to prove that a key is not set?
  We need to build a complete tree.

Complete Trees: Example with Ranges ii



- This is a way of building a complete Merkle tree, that is to say, a complete key-value tree.
- We will see other ways of building complete trees.

Merkle Trees Classification: Tree Updatability

- **Read-only:**
  - Tree data structure in which the data or set of elements it represents cannot be modified or updated once the tree is constructed.
  - In other words, the Merkle tree is static and immutable.
- **Append-only:**
  - Tree structure where data can only be appended to the existing dataset, but existing data cannot be modified or removed.
  - This means that once data is added to the tree, it becomes an immutable part of the tree's history.
- **Read-write:**
  - Tree data structure that allows not only the verification of data integrity but also the ability to modify or update the underlying data while preserving the tree's structure and maintaining the integrity verification features.

Merkle Trees for Zero Knowledge Proofs (ZKPs)

There are some additional aspects that need to be considered when building Merkle trees to provide ZK proofs:

a) Some operations that are efficient in the non-ZKP setting might be not efficient in the ZKP setting, which operates in a finite field.
   - An example are range checks, e.g. $x \geq 7$.
   - Another example are hash functions: while classical hash functions are bit-oriented, the ones used in ZKP are algebraic (e.g. Poseidon).

b) Operations must be deterministic and enough constrained:
   - In particular, we need to enforce that operations that update the tree are sound as well as the proofs of inclusion generated.

c) We have to adapt to the field being used by the cryptographic backend:
   - The size of the data that we want to store in the tree and the size of the underlying field used by the ZKP system might differ and we will have to adjust these sizes.

Techniques to Build Sparse Trees

- Partial tree construction:
  - Instead of building the entire tree, a sparse Merkle tree is constructed only for the keys and data elements that need to be represented.
  - This avoids the overhead of constructing and storing empty or unused portions of the tree.
- Path Compression:
  - Path compression involves collapsing adjacent internal nodes that have no active children.
  - This helps further reduce the tree's size and enhances efficiency.
  - This is the main technique used in prefix trees (also known as radix trees).
  - Note. Ethereum uses a type of radix tree called Patricia tree.
- Lazy evaluation:
  - Some sparse Merkle tree implementations use lazy evaluation to build the tree on-the-fly.
  - Internal nodes are computed and stored only when they are needed for verification, reducing the upfront computation and storage requirements.