SCROLL Scroll seamlessly extends Ethereum's capabilities through zero knowledge tech and EVM compatibility. The L2 network built by Ethereum devs for Ethereum devs.

Scroll Overview

Welcome to the Scroll docs!

Scroll is a security-focused scaling solution for Ethereum, using innovations in scaling design and zero knowledge proofs to build a new layer on Ethereum. The Scroll network is more accessible, more responsive, and can support more users at once than Ethereum alone, and if you've ever used or developed an application on Ethereum, you'll be right home on Scroll.

Want to try out the Scroll Sepolia testnet with free assets before using Scroll? Check out our User Guide.

What is Scroll building?

Scroll is building the technology to scale Ethereum.

While Ethereum is the leading blockchain network for powering decentralized applications, its popularity also brings higher costs, creating a barrier to adoption for the next wave of users and developers.

Leveraging cutting-edge research in zero knowledge proofs ("zk"), Scroll is building a Layer 2 rollup network on Ethereum. The team, in open-source collaboration with others in the Ethereum community, has created a "zkEVM" that allows for all activity on the network, which behaves just like Ethereum, to be secured by smart contracts *on* Ethereum. The network publishes all of the transactions to Ethereum, and the zkEVM creates and publishes cryptographic "proofs" that the Scroll network is following the rules of Ethereum.

Ultimately, Ethereum smart contracts verify that every transaction on Scroll is valid for these proofs, lending the network incredible security, decentralization, and censorship resistance. This level of security and scalability for Ethereum is only possible with recent breakthroughs in zero knowledge cryptography, blockchain protocol design, and hardware acceleration.

For more information on our architecture, see Scroll Architecture.

Scroll Architecture

Scroll chain consists of three layers:

- Settlement Layer: provides data availability and ordering for the canonical Scroll chain, verifies validity proofs, and allows users and dapps to send messages and assets between Ethereum and Scroll. We use Ethereum as the Settlement Layer and deploy the bridge and rollup contract onto the Ethereum.
- Sequencing Layer: contains a *Execution Node* that executes the transactions submitted to the Scroll sequencer and the transactions submitted to the L1 bridge contract and produces L2 blocks, and a *Rollup Node* that batches transactions, posts transaction data and block information to Ethereum for data availability, and submits validity proofs to Ethereum for finality.
- Proving Layer: consists of a pool of provers that are responsible for generating the zkEVM validity proofs that verify the correctness of L2 transactions, and a coordinator that dispatches the proving tasks to provers and relays the proofs to the Rollup Node to finalize on the Ethereum.

This section of the documentation provides comprehensive information on the Scroll protocol specification, bridging protocol, sequencer, and zkEVM circuit.

In the remainder of this section, L1 will refer to Ethereum, while L2 will refer to Scroll.

Can I use Scroll today?

Scroll mainnet on Ethereum is live! We also have a testnet running on Ethereum Sepolia, the Scroll Sepolia testnet. Although we have a comprehensive guide, if you're familiar with using Ethereum, you can get started in minutes:

- Visit our Bridge or Scroll Sepolia Bridge and connect your wallet
- Send tokens from Ethereum mainnet to Scroll (or use a Scroll Sepolia faucet)
- Test out Scroll Sepolia testnet dapp like our Uniswap Showcase or even Aave — just be sure to select the Scroll Sepolia network!

Scroll is its own Layer 2 network built on Ethereum (more specifically, a "zero knowledge rollup").

If you’re experienced in building on Ethereum, your code, dependencies, and tooling work with Scroll out of the box. This is possible because our network is compatible with EVM bytecode and designed to feel just like developing on Ethereum.

Use the table below to configure your Ethereum tools to the Scroll mainnet.

	Network Name	Scroll	Ethereum Mainnet
RPC URL		https://rpc.scroll.io/	https://eth.llamarpc.com
Chain ID		534352	1
Currency Symbol		ETH	ETH
Block Explorer URL		https://scrollscan.com/	https://etherscan.io

Hardhat

Modify your Hardhat config file `hardhat.config.ts` to point at the Scroll Sepolia Testnet public RPC.

```
...

const config: HardhatUserConfig = {
  ...
  networks: {
    scrollSepolia: {
      url: "https://sepolia-rpc.scroll.io/" || "",
      accounts:
        process.env.PRIVATE_KEY !== undefined ? [process.env.PRIVATE_KEY] : [],
    },
  },
};


…
```

Accounts and State

Accounts

Same as Ethereum, Scroll has two account types: Externally-owned account (EOA) and contract account that holds the smart contract and additional storages.

Scroll stores additional information of the contract bytecode in the account to facilitate the zkEVM circuit to prove the state transition more efficiently.

The account in Scroll contains the following fields:

nonce: A counter that indicates the number of transactions sent by the sender.

balance: The balance of ETH token in the account balance (unit in wei).

storageRoot: The root hash of the storage trie. Since Scroll uses the zkTrie for the storage trie, the storageRoot stores the Poseidon hash digest in a 256-bit integer.

codeHash: The Keccak hash digest of the contract bytecode.

PoseidonCodeHash (new field): The Poseidon hash digest of the contract bytecode in a 256-bit integer.

CodeSize (new field): The number of bytes in the contract bytecode.

State

The state of a blockchain is a collection of account data. The state trie encodes account data and their corresponding addresses to a Merkle tree data structure. The root of tree, or the state of the blockchain, is a cryptographic digest of all the account data contained in the tree.

Ethereum uses a data structure called Patricia Merkle Trie for both the state trie and the storage trie that stores the key-value entries stored in a smart contract. In Scroll, we replace the Patricia Merkle Trie by a more zk-friendly data structure, called zkTrie, for both state trie and storage trie. In the high level, the zkTrie data structure is a sparse binary Merkle tree with the Poseidon hash, a zk-friendly hash function. The zkTrie document describes more details about this data structure.

Transactions

A transaction is a cryptographically signed message that initiates a state transition to the chain state. It can be a simple balance transfer or invoking a smart contract, which in turn executes a program to alter the state.

Transaction Type

Currently, Scroll supports three types of transactions.

- Pre-EIP-155 Transaction: This is to support the Singleton Factory contract.
- Legacy Transaction (refer to EIP-155)
- `L1MessageTx` Typed Transaction (Type: `0x7E`): This is a new EIP-2718 transaction introduced in Scroll as described below. This transaction type is for transactions initiated on L1.

Note that EIP-2930 and EIP-1559 transaction type are not supported in Scroll currently. Scroll will bring back these two transaction types in the future.

L1 Message Transaction

We introduce a new type of transactions `L1MessageTx` for L1 initiated transactions. This type of transaction is initiated on the L1 bridge contract. Every time a new message is appended to the `L1MessageQueue` contract on L1, the L2 sequencer will create a corresponding `L1MessageTx` transaction to be included in the L2 blocks. Because the signature was already implicitly verified when users submitted the transaction on L1, `L1MessageTx` transactions don't have signature.

The `L1MessageTx` transaction type is `0x7E` and its payload is defined as follows.

```
type L1MessageTx struct {

    QueueIndex uint64    // The queue index of the message queue in L1 contract

    Gas        uint64    // Gas limit
```

```
        To          *common.Address // Cannot be nil, we do not allow contract creation
from L1

        Value       *big.Int

        Data        []byte

        Sender      common.Address

}
```

The RLP encoding of `L1MessageTx` transactions follows the EIP-2718 rule `TransactionType || TransactionPayload`, where `TransactionType` is `0x7E` and `TransactionPayload = RLP(L1MessageTx)`.

Two noticeable behaviors of the `L1MessageTx` transactions:

- The `QueueIndex` in the transaction is the queue index in the L1 message queue, different from the `Nonce` of the `Sender` account. But the sender's `Nonce` will still increases by 1 after the transaction.
- This type of transactions doesn't pay L2 fee or L1 fee. The L2 fee is already paid when users submit transactions on L1. The L1 Fee isn't charged because the data of `L1MessageTx` transactions is already available in the L1 bridge contract.

Transaction Life Cycle

The transaction life cycle in the Scroll contains the following three phases:

1. Confirmed: Users submits a transaction to either the L1 bridge contract or L2 sequencer. The transaction becomes `Confirmed` after it gets executed and included in a L2 block.
2. Committed: The transactions are included in a batch and a *commit transaction* that contains the data of this batch is submitted to L1. After the commit transaction is finalized in the L1 blocks, the transactions in this batch becomes `Committed`.
3. Finalized: The validity proof of this batch is generated and verified on the L1. After the *finalize transaction* is finalized on L1, the status of the transaction is `Finalized` and becomes a canonical part of the Scroll L2 chain.

Submit Transactions

There are two entry points where users can submit transactions to Scroll.

First, users can directly submit transactions to L2 sequencers. To do so, users just need to configure their wallets and connect to the Scroll RPC endpoint. The sequencer validates the transaction and stores it in its local transaction pool.

Second, deposit and enforced transactions are originated on L1. Scroll L1 bridge contract provides three entry points for users and smart contracts to send transactions from the L1. All messages sent through these three entry points will be appended to the `L1MessageQueue` contract.

- The `ScrollGatewayRouter` contract and several standard token gateways allow users and contracts to deposit standard tokens to L2. See more details in the Deposit Token Gateways.
- The `L1ScrollMessenger` contract allows users and contracts to send arbitrary messages to L2. See more details in the Sending Arbitrary Messages.
- The `EnforcedTxGateway` contract allows EOAs to initiate an enforced transaction from the same address to withdraw tokens or call other contracts on L2. See more details in the Sending Enforced Transaction.

The Scroll sequencer periodically starts a new mining job. It pulls the L1 messages from the `L1MessageQueue` contract and transactions in the L2 mempool and seals a block. Once a transaction is included in a L2 block, its status becomes `Confirmed`.

Commit Transaction Data

The rollup node collects new L2 blocks and packs them into chunks and batches (see more details in Transaction Batching). Periodically it sends a *Commit Transaction* that posts the data of a batch of transactions to the L1 `ScrollChain` contract. After the Commit Transaction is finalized in a L1 block, the status of the transactions in this batch becomes `Committed`. At this time, users can reconstruct L2 state themselves completely based on the committed data from the L1 contract.

Finalize Transactions

After the validity proof is generated, the rollup node sends a *Finalize Transaction* including the validity proof and the new state root after this batch for onchain verification. Once the Finalize Transaction succeeds and confirmed in a L1 block, the status of the L2 transactions in this batch becomes `Finalized`. The new state root can be used by third parties trustlessly.

Transaction Batching

In Scroll, the transactions are batched in multiple tiers.

1. A group of ordered transactions are packed into a block.
2. A series of contiguous blocks are grouped into a chunk. The chunk is the base unit for proof generation of the zkEVM circuit.
3. A series of contiguous chunks are grouped into a batch. The batch is the base unit for data commitment and proof verification on the L1. The proof for a batch, or a *batch proof*, is an aggregated proof of the chunk proofs in this batch.

The goal for this multi-layer batching schema is to reduce the gas cost of onchain data commitment and proof verification. This approach increases the granularity of the rollup units on L1 while takes the fixed circuit capacity into consideration. As a result, batching reduces the data to be stored in the contract and amortizes the proof verification cost to more L2 transactions.

Once a chunk is created, a corresponding chunk proving task will be generated and sent to a zkEVM prover. Upon the creation of a new batch, two subsequent actions occur: (a) the rollup node commits the transaction data and block information from this batch to the L1 contract, and (b) a batch proving task to aggregate chunk proofs is dispatched to an aggregator prover. The standards for proposing a chunk and a batch are detailed in the Rollup Node.

Scroll Contracts

Rollup

- L1 Rollup (Scroll Chain): `0xa13BAF47339d63B743e7Da8741db5456DAc1E556`

ETH and ERC20 Bridge

- L1 ERC20 Gateway Router: `0xF8B1378579659D8F7EE5f3C929c2f3E332E41Fd6`
- L2 ERC20 Gateway Router: `0x4C0926FF5252A435FD19e10ED15e5a249Ba19d79`

Advanced Bridge Contracts

- Scroll Messenger
    - L1 Messenger: `0x6774Bcbd5ceCeF1336b5300fb5186a12DDD8b367`
    - L2 Messenger: `0x781e90f1c8Fc4611c9b7497C3B47F99Ef6969CbC`
- ETH Bridge
    - L1 ETH Gateway: `0x7F2b8C31F88B6006c382775eea88297Ec1e3E905`
    - L2 ETH Gateway: `0x6EA73e05AdC79974B931123675ea8F78FfdacDF0`
    - L1 WETH Gateway: `0x7AC440cAe8EB6328de4fA621163a792c1EA9D4fE`

    - L2 WETH Gateway: `0x7003E7B7186f0E6601203b99F7B8DECBfA391cf9`
- ERC20 Bridge
    - L1 ERC20 Standard Gateway: `0xD8A791fE2bE73eb6E6cF1eb0cb3F36adC9B3F8f9`
    - L2 ERC20 Standard Gateway: `0xE2b4795039517653c5Ae8C2A9BFdd783b48f447A`
    - L1 ERC20 Custom Gateway: `0xb2b10a289A229415a124EFDeF310C10cb004B6ff`
    - L2 ERC20 Custom Gateway: `0x64CCBE37c9A82D85A1F2E74649b7A42923067988`
- ERC721 Bridge
    - L1 ERC721 Gateway: `0x6260aF48e8948617b8FA17F4e5CEa2d21D21554B`
    - L2 ERC721 Gateway: `0x7bC08E1c04fb41d75F1410363F0c5746Eae80582`
- ERC1155 Bridge
    - L1 ERC1155 Gateway: `0xb94f7F6ABcb811c5Ac709dE14E37590fcCd975B6`
    - L2 ERC1155 Gateway: `0x62597Cc19703aF10B58feF87B0d5D29eFE263bcc`
- Gas Oracle
    - L2 Gas Oracle (deployed on Mainnet): `0x987e300fDfb06093859358522a79098848C33852`

L2 Predeploys

- Message Queue: `0x5300000000000000000000000000000000000000`
- Gas Price Oracle: `0x5300000000000000000000000000000000000002`
- Whitelist: `0x5300000000000000000000000000000000000003`
- WETH L2: `0x5300000000000000000000000000000000000004`
- Transaction Fee Vault: `0x5300000000000000000000000000000000000005`

Protocols on Scroll Mainnet

Uniswap v3

- Main Contracts
    - Core Factory: `0x70C62C8b8e801124A4Aa81ce07b637A3e83cb919`
    - NFT Position Manager: `0xB39002E4033b162fAc607fc3471E205FA2aE5967`
    - Router: `0xfc30937f5cDe93Df8d48aCAF7e6f5D8D8A31F636`
- Additional Contracts
    - multicall2Address: `0xC1D2e074C38FdD5CA965000668420C80316F0915`
    - proxyAdminAddress: `0x1E6dcAb806A42055098f23E2B3ac72D6E195F967`
    - tickLensAddress: `0x85780e12e90D2a684eB8E7404c985b5B5c8ce7E9`
    - nftDescriptorLibraryAddressV1_3_0: `0xAeE9c206ba89F3DA25EEe4636208519e0B86965B`

- nonfungibleTokenPositionDescriptorAddressV1_3_0: `0xACcf12204b7591B2ECCEFe737440B0f53748B191`
- descriptorProxyAddress: `0x675DD953225D296A44790dC1390a1E7eF378f464`
- v3MigratorAddress: `0xF00577B5Dd0DA227298E954Ed11356F264Cf93d4`
- v3StakerAddress: `0xFdFbE973c9ecB036Ecfb7af697FcACe789D3f928`
- quoterV2Address: `0x2566e082Cb1656d22BCbe5644F5b997D194b5299`

Additional Useful Contracts

- Multicall3: `0xcA11bde05977b3631167028862bE2a173976CA11`

Scroll Messenger Cross-chain Interaction

In this example, we will launch a dummy smart contract on either Sepolia or Scroll and interact with it from the opposite chain. We will be using the `ScrollMessenger` that is deployed on both Sepolia and Scroll.

Deploying the Contracts

Target Smart Contract

Let’s start by deploying the target smart contract. We will use the Greeter contract for this example, but you can use any other contract. Deploy it to either Sepolia or Scroll. On Scroll, L1 and L2 use the same API, so it’s up to you.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.16;

// This Greeter contract will be interacted with through the ScrollMessenger across the bridge

```
contract Greeter {

  string public greeting = "Hello World!";



  // This function will be called by executeFunctionCrosschain on the Operator Smart Contract

  function setGreeting(string memory greeting_) public {

    greeting = greeting_;

  }

}
```

We will now execute `setGreeting` in a cross-chain way.

Operator Smart Contract

Switch to the other chain and deploy the `GreeterOperator`. So, if you deployed the `Greeter` contract on L1, deploy the `GreeterOperator` on L2 or vice versa.


```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.16;
```

```
// The Scroll Messenger interface is the same on both L1 and L2, it allows sending
cross-chain transactions

// Let's import it directly from the Scroll Contracts library

import "@scroll-tech/contracts@0.1.0/libraries/IScrollMessenger.sol";



// The GreeterOperator is capable of executing the Greeter function through the bridge

contract GreeterOperator {

  // This function will execute setGreeting on the Greeter contract

  function executeFunctionCrosschain(

    address scrollMessengerAddress,

    address targetAddress,

    uint256 value,

    string memory greeting,

    uint32 gasLimit
```

```
  ) public payable {

    IScrollMessenger scrollMessenger = IScrollMessenger(scrollMessengerAddress);

    // sendMessage is able to execute any function by encoding the abi using the
encodeWithSignature function

    scrollMessenger.sendMessage{ value: msg.value }(

      targetAddress,

      value,

      abi.encodeWithSignature("setGreeting(string)", greeting),

      gasLimit,

      msg.sender

    );

  }

}
```

Calling a Cross-chain Function

We pass the message by executing `executeFunctionCrosschain` and passing the following parameters:

- `scrollMessengerAddress`: This will depend on where you deployed the `GreeterOperator` contract.
    - If you deployed it on Sepolia use `0x50c7d3e7f7c656493D1D76aaa1a836CedfCBB16A`. If you deployed on Scroll use `0xBa50f5340FB9F3Bd074bD638c9BE13eCB36E603d`.
- `targetAddress`: The address of the `Greeter` contract on the opposite chain.
- `value`: In this case, it is `0` because the `setGreeting`is not payable.
- `greeting`: This is the parameter that will be sent through the message. Try passing `"This message was crosschain!"`
- `gasLimit`:
    - If you are sending the message from L1 to L2, around `5000` gas limit should be more than enough.
    - If you are sending the message from L2 to L1, pass `0`, as the transaction be completed by executing an additional transaction on L1.

Relay the Message when sending from L2 to L1

When a transaction is passed from L2 to L1, an additional "execute withdrawal transaction" must be sent on L1. To do this, you must call `relayMessageWithProof` on the L1 Scroll Messenger contract from an EOA wallet.

You can do this directly on Etherscan Sepolia. To do so, you will need to pass a Merkle inclusion proof for the bridged transaction and other parameters. You'll query these using the Scroll Bridge API.

We're finalizing the API specifics, but for now, fetch or curl the following endpoint:

curl "https://sepolia-api-bridge.scroll.io/api/claimable?page_size=10&page=1&address=GREETER_OPERATOR_ADDRESS_ON_L2"

Replace `GREETER_OPERATOR_ADDRESS_ON_L2` with your GreeterOperator contract address as launched on L2. Read more about Execute Withdraw transactions in the Scroll Messenger article.

Rollup Process

This document describes the rollup process in Scroll.

Workflow

The figure illustrates the rollup workflow. The L2 sequencer contains three modules:

- Sync service subscribes to the event issued from the L1 bridge contract. Once it detects any newly appended messages to the L1 inbox, the Sync Service will generate a new `L1MessageTx` transaction accordingly and add it to the local L1 transaction queue.
- Mempool collects the transactions that are directly submitted to the L2 sequencer.
- Executor pulls the transactions from both the L1 transaction queue and L2 mempool, executes them to construct a new L2 block.

The rollup node contains three modules:

- Relayer submits the commit transactions and finalize transactions to the rollup contract for data availability and finality.
- Chunk Proposer and Batch Proposer proposes new chunks and new batches following the constraints described in the Transaction Batching.

The rollup process can be broken down into three phases: transaction execution, batching and data commitment, and proof generation and finalization.

Phase 1. Transaction Execution

1. Users submit transactions to L1 bridge contract or L2 sequencers.
2. The Sync Service in the L2 sequencer fetches the latest appended L1 transactions from the bridge contract.
3. The L2 sequencer processes the transactions from both the L1 message queue and the L2 mempool to construct L2 blocks.

Phase 2. Batching and Data Commitment

4. The rollup node monitors the latest L2 blocks and fetches the transaction data.
5. If the criterion (described in the Transaction Batching) are met, the rollup node proposes a new chunk or a batch and writes it to the database. Otherwise, the rollup node keeps waiting for more blocks or chunks.
6. Once a new batch is created, the rollup relayer collects the transaction data in this batch and submits a Commit Transaction to the rollup contract for data availability.

Phase 3. Proof Generation and Finalization

7. Once the coordinator polls a new chunk or batch fromm the database:
    - Upon a new chunk, the coordinator will query the execution traces of all blocks in this chunk from the L2 sequencer and then send a chunk proving task to a randomly selected zkEVM prover.
    - Upon a new batch, the coordinator will collect the chunk proofs of all chunks in this batch from the database and dispatch a batch proving task to a randomly selected aggregator prover.
8. Upon the coordinator receives a chunk or batch proofs from a prover, it will write the proof to the database.
9. Once the rollup relayer polls a new batch proof from the database, it will submit a Finalize Transaction to the rollup contract to verify the proof.

Commit Transaction

The Commit Transaction submits the block information and transaction data to L1 for data availability. The transaction includes the parent batch header, chunk data, and a bitmap of skipped L1 message. The parent batch header designates the previous batch that this batch links to. The parent batch must be committed before; otherwise the transaction will be reverted. See the `commitBatch` function signature below.

```
function commitBatch(
    uint8 version,
    bytes calldata parentBatchHeader,
    bytes[] memory chunks,
    bytes calldata skippedL1MessageBitmap
) external override OnlySequencer
```

After the `commitBatch` function verifies the parent batch is committed before, it constructs the batch header of the batch and stores the hash of the batch header in the `ScrollChain` contract.


```
mapping(uint256 => bytes32) public committedBatches;
```

The encoding of batch header and chunk data are described in the Codec section. Most fields in the batch header are straight-forward to derive from the chunk data. One important field to note is `dataHash` that will become part of the public input to verify the validity proof. Assuming that a batch contains `n` chunks, its `dataHash` is computed as follows


```
batch.dataHash := keccak(chunk[0].dataHash || ... || chunk[n-1].dataHash)
```

Assuming that a chunk contains `k` blocks, its `dataHash` is computed as follows


```
chunk.dataHash := keccak(blockContext[0] || ... || blockContext[k-1] ||
                         block[0].l1TxHashes || block[0].l2TxHashes || ... ||
                         block[k-1].l1TxHashes || block[k-1].l2TxHashes)
```

, where `block.l1TxHashes` are the concatenated transaction hashes of the L1 transactions in this block and `block.l2TxHashes` are the concatenated transaction hashes of the L2 transactions in this block. Note that the transaction hashes of L1 transactions are not uploaded by the rollup node, but instead directly loaded from the `L1MessageQueue` contract given the index range of included L1 messages in this block. The L2 transaction hashes are calculated from the RLP-encoded bytes in the `l2Transactions` field in the `Chunk`.

In addition, the `commitBatch` function contains a bitmap of skipped L1 messages. Unfortunately, this is due to the problem of proof overflow. If the L1 transaction corresponding to a L1 message exceeds the circuit capacity limit, we won't be able to generate a valid proof for this transaction and thus cannot finalize it on L1. Scroll is working actively to eliminate the proof overflow problem through upgrades to our proving system.

Finalize Transaction