---

title: Zero-Knowledge rollups description: An introduction to zero-knowledge rollups—a scaling solution used by the Ethereum community. lang: en

---

Zero-knowledge rollups (ZK-rollups) are layer 2scaling solutions that increase throughput on Ethereum Mainnet by moving computation and state-storage off-chain. ZK-rollups can process thousands of transactions in a batch and then only post some minimal summary data to Mainnet. This summary data defines the changes that should be made to the Ethereum state and some cryptographic proof that those changes are correct.

Prerequisites {#prerequisites}

You should have read and understood our page onEthereum scaling and layer 2.

What are zero-knowledge rollups? {#what-are-zk-rollups}

Zero-knowledge rollups (ZK-rollups) bundle (or 'roll up') transactions into batches that are executed off-chain. Off-chain computation reduces the amount of data that has to be posted to the blockchain. ZK-rollup operators submit a summary of the changes required to represent all the transactions in a batch rather than sending each transaction individually. They also produce validity proofs to prove the correctness of their changes.

The ZK-rollup's state is maintained by a smart contract deployed on the Ethereum network. To update this state, ZK-rollup nodes must submit a validity proof for verification. As mentioned, the validity proof is a cryptographic assurance that the state-change proposed by the rollup is really the result of executing the given batch of transactions. This means that ZK-rollups only need to provide validity proofs to finalize transactions on Ethereum instead of posting all transaction data on-chain like optimistic rollups.

There are no delays when moving funds from a ZK-rollup to Ethereum because exit transactions are executed once the ZK-rollup contract verifies the validity proof. Conversely, withdrawing funds from optimistic rollups is subject to a delay to allow anyone to challenge the exit transaction with a fraud proof.

ZK-rollups write transactions to Ethereum as `calldata`. `calldata` is where data that is included in external calls to smart contract functions gets stored. Information in `calldata` is published on the blockchain, allowing anyone to reconstruct the rollup's state independently. ZK-rollups use compression techniques to reduce transaction data—for example, accounts are represented by an index rather than an address, which saves 28 bytes of data. On-chain data publication is a significant cost for rollups, so data compression can reduce fees for users.

How do ZK-rollups interact with Ethereum? {#zk-rollups-and-ethereum}

A ZK-rollup chain is an off-chain protocol that operates on top of the Ethereum blockchain and is managed by on-chain Ethereum smart contracts. ZK-rollups execute transactions outside of Mainnet, but periodically commit off-chain transaction batches to an on-chain rollup contract. This transaction record is immutable, much like the Ethereum blockchain, and forms the ZK-rollup chain.

The ZK-rollup's core architecture is made up of the following components:

1. **On-chain contracts**: As mentioned, the ZK-rollup protocol is controlled by smart contracts running on Ethereum. This includes the main contract which stores rollup blocks, tracks deposits, and monitors state updates. Another on-chain contract (the verifier contract) verifies zero-knowledge proofs submitted by block producers. Thus, Ethereum serves as the base layer or "layer 1" for the ZK-rollup.

2. **Off-chain virtual machine (VM)**: While the ZK-rollup protocol lives on Ethereum, transaction execution and state storage happen on a separate virtual machine independent of the EVM. This off-chain VM is the execution environment for transactions on the ZK-rollup and serves as the secondary layer or "layer 2" for the ZK-rollup protocol. Validity proofs verified on Ethereum Mainnet guarantee the correctness of state transitions in the off-chain VM.

ZK-rollups are "hybrid scaling solutions"—off-chain protocols that operate independently but derive security from Ethereum. Specifically, the Ethereum network enforces the validity of state updates on the ZK-rollup and guarantees the availability of data behind every update to the rollup's state. As a result, ZK-rollups are considerably safer than pure off-chain scaling solutions, such as sidechains, which are responsible for their security properties, orvalidiums, which also verify transactions on Ethereum with validity proofs, but store transaction data elsewhere.

ZK-rollups rely on the main Ethereum protocol for the following:

Data availability {#data-availability}

ZK-rollups publish state data for every transaction processed off-chain to Ethereum. With this data, it is possible for individuals or businesses to reproduce the rollup's state and validate the chain themselves. Ethereum makes this data available to all participants of the network as `calldata`.

ZK-rollups don't need to publish much transaction data on-chain because validity proofs already verify the authenticity of state transitions. Nevertheless, storing data on-chain is still important because it allows permissionless, independent verification of the L2 chain's state which in turn allows anyone to submit batches of transactions, preventing malicious operators from censoring or freezing the chain.

On-chain is required for users to interact with the rollup. Without access to state data users cannot query their account balance or initiate transactions (e.g., withdrawals) that rely on state information.

Transaction finality {#transaction-finality}

Ethereum acts as a settlement layer for ZK-rollups: L2 transactions are finalized only if the L1 contract accepts the validity proof. This eliminates the risk of malicious operators corrupting the chain (e.g., stealing rollup funds) since every transaction must be approved on Mainnet. Also, Ethereum guarantees that user operations cannot be reversed once finalized on L1.

Censorship resistance {#censorship-resistance}

Most ZK-rollups use a "supernode" (the operator) to execute transactions, produce batches, and submit blocks to L1. While this ensures efficiency, it increases the risk of censorship: malicious ZK-rollup operators can censor users by refusing to include their transactions in batches.

As a security measure, ZK-rollups allow users to submit transactions directly to the rollup contract on Mainnet if they think they are being censored by the operator. This allows users to force an exit from the ZK-rollup to Ethereum without having to rely on the operator's permission.

How do ZK-rollups work? {#how-do-zk-rollups-work}

Transactions {#transactions}

Users in the ZK-rollup sign transactions and submit to L2 operators for processing and inclusion in the next batch. In some cases, the operator is a centralized entity, called a sequencer, who executes transactions, aggregates them into batches, and submits to L1. The sequencer in this system is the only entity allowed to produce L2 blocks and add rollup transactions to the ZK-rollup contract.

Other ZK-rollups may rotate the operator role by using a[proof-of-stake](#) validator set. Prospective operators deposit funds in the rollup contract, with the size of each stake influencing the staker's chances of getting selected to produce the next rollup batch. The operator's stake can be slashed if they act maliciously, which incentivizes them to post valid blocks.

How ZK-rollups publish transaction data on Ethereum {#how-zk-rollups-publish-transaction-data-on-ethereum}

As explained, transaction data is published on Ethereum as`calldata`. `calldata` is a data area in a smart contract used to pass arguments to a function and behaves similarly to [memory](#). While `calldata` isn't stored as part of Ethereum's state, it persists on-chain as part of the Ethereum chain's [history logs](#). `calldata` does not affect Ethereum's state, making it a cheap way to store data on-chain.

The `calldata` keyword often identifies the smart contract method being called by a transaction and holds inputs to the method in the form of an arbitrary sequence of bytes. ZK-rollups use `calldata` to publish compressed transaction data on-chain; the rollup operator simply adds a new batch by calling the required function in the rollup contract and passes the compressed data as function arguments. This helps reduce costs for users since a large part of rollup fees go toward storing transaction data on-chain.

State commitments {#state-commitments}

The ZK-rollup's state, which includes L2 accounts and balances, is represented as a[Merkle tree](#). A cryptographic hash of the

Merkle tree's root (Merkle root) is stored in the on-chain contract, allowing the rollup protocol to track changes in the state of the ZK-rollup.

The rollup transitions to a new state after the execution of a new set of transactions. The operator who initiated the state transition is required to compute a new state root and submit to the on-chain contract. If the validity proof associated with the batch is authenticated by the verifier contract, the new Merkle root becomes the ZK-rollup's canonical state root.

Besides computing state roots, the ZK-rollup operator also creates a batch root—the root of a Merkle tree comprising all transactions in a batch. When a new batch is submitted, the rollup contract stores the batch root, allowing users to prove a transaction (e.g., a withdrawal request) was included in the batch. Users will have to provide transaction details, the batch root, and a Merkle proof showing the inclusion path.

Validity proofs {#validity-proofs}

The new state root that the ZK-rollup operator submits to the L1 contract is the result of updates to the rollup's state. Say Alice sends 10 tokens to Bob, the operator simply decreases Alice's balance by 10 and increments Bob's balance by 10. The operator then hashes the updated account data, rebuilds the rollup's Merkle tree, and submits the new Merkle root to the on-chain contract.

But the rollup contract won't automatically accept the proposed state commitment until the operator proves the new Merkle root resulted from correct updates to the rollup's state. The ZK-rollup operator does this by producing a validity proof, a succinct cryptographic commitment verifying the correctness of batched transactions.

Validity proofs allow parties to prove the correctness of a statement without revealing the statement itself—hence, they are also called zero-knowledge proofs. ZK-rollups use validity proofs to confirm the correctness of off-chain state transitions without having to re-execute transactions on Ethereum. These proofs can come in the form of a ZK-SNARK (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) or ZK-STARK (Zero-Knowledge Scalable Transparent Argument of Knowledge).

Both SNARKs and STARKs help attest to the integrity of off-chain computation in ZK-rollups, although each proof type has distinctive features.

ZK-SNARKs

For the ZK-SNARK protocol to work, creating a Common Reference String (CRS) is necessary: the CRS provides public parameters for proving and verifying validity proofs. The security of the proving system depends on the CRS setup; if information used to create public parameters fall into the possession of malicious actors they may be able to generate false validity proofs.

Some ZK-rollups attempt to solve this problem by using amulti-party computation ceremony (MPC), involving trusted individuals, to generate public parameters for the ZK-SNARK circuit. Each party contributes some randomness (called "toxic waste") to the construct the CRS, which they must destroy immediately.

Trusted setups are used because they increase the security of the CRS setup. As long as one honest participant destroys their input, the security of the ZK-SNARK system is guaranteed. Still, this approach requires trusting those involved to delete their sampled randomness and not undermine the system's security guarantees.

Trust assumptions aside, ZK-SNARKs are popular for their small proof sizes and constant-time verification. As proof verification on L1 constitutes the larger cost of operating a ZK-rollup, L2s use ZK-SNARKs to generate proofs that can be verified quickly and cheaply on Mainnet.

ZK-STARKs

Like ZK-SNARKs, ZK-STARKs prove the validity of off-chain computation without revealing the inputs. However, ZK-STARKs are considered an improvement on ZK-SNARKs because of their scalability and transparency.

ZK-STARKs are 'transparent', as they can work without the trusted setup of a Common Reference String (CRS). Instead, ZK-STARKs rely on publicly verifiable randomness to set up parameters for generating and verifying proofs.

ZK-STARKs also provide more scalability because the time needed to prove and verify validity proofs increases *quasilinearly* in relation to the complexity of the underlying computation. With ZK-SNARKs, proving and verification times scale *linearly* in relation to the size of the underlying computation. This means ZK-STARKs require less time than ZK-SNARKs for proving and verifying when large datasets are involved, making them useful for high-volume applications.

ZK-STARKs are also secure against quantum computers, while the Elliptic Curve Cryptography (ECC) used in ZK-SNARKs is

widely believed to be susceptible to quantum computing attacks. The downside to ZK-STARKs is that they produce larger proof sizes, which are more expensive to verify on Ethereum.

How do validity proofs work in ZK-rollups? {#validity-proofs-in-zk-rollups}

Proof generation

Before accepting transactions, the operator will perform the usual checks. This includes confirming that:

- The sender and receiver accounts are part of the state tree.
- The sender has enough funds to process the transaction.
- The transaction is correct and matches the sender's public key on the rollup.
- The sender's nonce is correct, etc.

Once the ZK-rollup node has enough transactions, it aggregates them into a batch and compiles inputs for the proving circuit to compile into a succinct ZK-proof. This includes:

- A Merkle tree root comprising all the transactions in the batch.
- Merkle proofs for transactions to prove inclusion in the batch.
- Merkle proofs for each sender-receiver pair in transactions to prove those accounts are part of the rollup's state tree.
- A set of intermediate state roots, derived from updating the state root after applying state updates for each transaction (i.e., decreasing sender accounts and increasing receiver accounts).

The proving circuit computes the validity proof by "looping" over each transaction and performing the same checks the operator completed before processing the transaction. First, it verifies the sender's account is part of the existing state root using the provided Merkle proof. Then it reduces the sender's balance, increases their nonce, hashes the updated account data and combines it with the Merkle proof to generate a new Merkle root.

This Merkle root reflects the sole change in the ZK-rollup's state: a change in the sender's balance and nonce. This is possible because the Merkle proof used to prove the account's existence is used to derive the new state root.

The proving circuit performs the same process on the receiver's account. It checks if the receiver's account exists under the intermediate state root (using the Merkle proof), increases their balance, re-hashes the account data and combines it with the Merkle proof to generate a new state root.

The process repeats for every transaction; each "loop" creates a new state root from updating the sender's account and a subsequent new root from updating the receiver's account. As explained, every update to the state root represents one part of the rollup's state tree changing.

The ZK-proving circuit iterates over the entire transaction batch, verifying the sequence of updates that result in a final state root after the last transaction is executed. The last Merkle root computed becomes the newest canonical state root of the ZK-rollup.

Proof verification

After the proving circuit verifies the correctness of state updates, the L2 operator submits the computed validity proof to the verifier contract on L1. The contract's verification circuit verifies the proof's validity and also checks public inputs that form part of the proof:

- **Pre-state root**: The ZK-rollup's old state root (i.e., before the batched transactions were executed), reflecting the L2 chain's last known valid state.
- **Post-state root**: The ZK-rollup's new state root (i.e., after the execution of batched transactions), reflecting the L2 chain's newest state. The post-state root is the final root derived after applying state updates in the proving circuit.
- **Batch root**: The Merkle root of the batch, derived by *merklizing* transactions in the batch and hashing the tree's root.
- **Transaction inputs**: Data associated with the transactions executed as part of the submitted batch.

If the proof satisfies the circuit (i.e., it is valid), it means that there exists a sequence of valid transactions that transition the rollup from the previous state (cryptographically fingerprinted by the pre-state root) to a new state (cryptographically fingerprinted by the post-state root). If the pre-state root matches the root stored in the rollup contract, and the proof is valid, the rollup contract takes the post-state root from the proof and updates its state tree to reflect the rollup's changed state.

Entries and exits {#entries-and-exits}

Users enter the ZK-rollup by depositing tokens in the rollup's contract deployed on the L1 chain. This transaction is queued up since only operators can submit transactions to the rollup contract.

If the pending deposit queue starts filling up, the ZK-rollup operator will take the deposit transactions and submit them to the rollup contract. Once the user's funds are in the rollup, they can start transacting by sending transactions to the operator for processing. Users can verify balances on the rollup by hashing their account data, sending the hash to the rollup contract, and providing a Merkle proof to verify against the current state root.

Withdrawing from a ZK-rollup to L1 is straightforward. The user initiates the exit transaction by sending their assets on the rollup to a specified account for burning. If the operator includes the transaction in the next batch, the user can submit a withdrawal request to the on-chain contract. This withdrawal request will include the following:

- Merkle proof proving the inclusion of the user's transaction to the burn account in a transaction batch
- Transaction data
- Batch root
- L1 address to receive deposited funds

The rollup contract hashes the transaction data, checks if the batch root exists, and uses the Merkle proof to check if the transaction hash is part of the batch root. Afterward, the contract executes the exit transaction and sends funds to the user's chosen address on L1.

ZK-rollups and EVM compatibility {#zk-rollups-and-evm-compatibility}

Unlike optimistic rollups, ZK-rollups are not readily compatible with theEthereum Virtual Machine (EVM). Proving general-purpose EVM computation in circuits is more difficult and resource-intensive than proving simple computations (like the token transfer described previously).

However, advances in zero-knowledge technologyare igniting renewed interest in wrapping EVM computation in zero-knowledge proofs. These efforts are geared towards creating a zero-knowledge EVM (zkEVM) implementation that can efficiently verify the correctness of program execution. A zkEVM recreates existing EVM opcodes for proving/verification in circuits, allowing to execute smart contracts.

Like the EVM, a zkEVM transitions between states after computation is performed on some inputs. The difference is that the zkEVM also creates zero-knowledge proofs to verify the correctness of every step in the program’s execution. Validity proofs could verify the correctness of operations that touch the VM’s state (memory, stack, storage) and the computation itself (i.e., did the operation call the right opcodes and execute them correctly?).

The introduction of EVM-compatible ZK-rollups is expected to help developers leverage the scalability and security guarantees of zero-knowledge proofs. More importantly, compatibility with native Ethereum infrastructure means developers can build ZK-friendly dapps using familiar (and battle-tested) tooling and languages.

How do ZK-rollup fees work? {#how-do-zk-rollup-fees-work}

How much users pay for transactions on ZK-rollups is dependent on the gas fee, just like on Ethereum Mainnet. However, gas fees work differently on L2 and are influenced by the following costs:

1. **State write**: There is a fixed cost for writing to Ethereum’s state (i.e., submitting a transaction on the Ethereum blockchain). ZK-rollups reduce this cost by batching transactions and spreading fixed costs across multiple users.
2. **Data publication**: ZK-rollups publish state data for every transaction to Ethereum as`calldata`. `calldata` costs are currently governed by EIP-1559, which stipulates a cost of 16 gas for non-zero bytes and 4 gas for zero bytes of`calldata`, respectively. The cost paid on each transaction is influenced by how much `calldata` needs to be posted on-chain for it.
3. **L2 operator fees**: This is the amount paid to the rollup operator as compensation for computational costs incurred in processing transactions, much like miner fees on Ethereum.

4. **Proof generation and verification**: ZK-rollup operators must produce validity proofs for transaction batches, which is resource-intensive. Verifying zero-knowledge proofs on Mainnet also costs gas (~ 500,000 gas).

Apart from batching transactions, ZK-rollups reduce fees for users by compressing transaction data. You cansee a real-time overview of how it costs to use Ethereum ZK-rollups.

How do ZK-rollups scale Ethereum? {#scaling-ethereum-with-zk-rollups}

Transaction data compression {#transaction-data-compression}

ZK-rollups extend the throughput on Ethereum's base layer by taking computation off-chain, but the real boost for scaling comes from compressing transaction data. Ethereum's block size limits the data each block can hold and, by extension, the number of transactions processed per block. By compressing transaction-related data, ZK-rollups significantly increase the number of transactions processed per block.

ZK-rollups can compress transaction data better than optimistic rollups since they don't have to post all the data required to validate each transaction. They only have to post the minimal data required to rebuild the latest state of accounts and balances on the rollup.

Recursive proofs {#recursive-proofs}

An advantage of zero-knowledge proofs is that proofs can verify other proofs. For example, a single ZK-SNARK can verify other ZK-SNARKs. Such "proof-of-proofs" are called recursive proofs and dramatically increase throughput on ZK-rollups.

Currently, validity proofs are generated on a block-by-block basis and submitted to the L1 contract for verification. However, verifying single block proofs limits the throughput that ZK-rollups can achieve since only one block can be finalized when the operator submits a proof.

Recursive proofs, however, make it possible to finalize several blocks with one validity proof. This is because the proving circuit recursively aggregates multiple block proofs until one final proof is created. The L2 operator submits this recursive proof, and if the contract accepts it, all the relevant blocks will be finalized instantly. With recursive proofs, the number of ZK-rollup transactions that can be finalized on Ethereum at intervals increases.

Pros and cons of ZK-rollups {#zk-rollups-pros-and-cons}

| Pros | Cons | | ---------- | ---------- | | Validity proofs ensure correctness of off-chain transactions and prevent operators from executing invalid state transitions. | The cost associated with computing and verifying validity proofs is substantial and can increase fees for rollup users. | | Offers faster transaction finality as state updates are approved once validity proofs are verified on L1. | Building EVM-compatible ZK-rollups is difficult due to complexity of zero-knowledge technology. | | Relies on trustless cryptographic mechanisms for security, not the honesty of incentivized actors as with optimistic rollups. | Producing validity proofs requires specialized hardware, which may encourage centralized control of the chain by a few parties. | | Stores data needed to recover the off-chain state on L1, which guarantees security, censorship-resistance, and decentralization. | Centralized operators (sequencers) can influence the ordering of transactions. | | Users benefit from greater capital efficiency and can withdraw funds from L2 without delays. | Hardware requirements may reduce the number of participants that can force the chain to make progress, increasing the risk of malicious operators freezing the rollup's state and censoring users. | | Doesn't depend on liveness assumptions and users don't have to validate the chain to protect their funds. | Some proving systems (e.g., ZK-SNARK) require a trusted setup which, if mishandled, could potentially compromise a ZK-rollup's security model. | | Better data compression can help reduce the costs of publishing `calldata` on Ethereum and minimize rollup fees for users. | |

A visual explanation of ZK-rollups {#zk-video}

Watch Finematics explain ZK-rollups:

Use ZK-rollups {#use-zk-rollups}

Multiple implementations of ZK-rollups exist that you can integrate into your dapps:

Who is working on a zkEVM? {#zkevm-projects}

Projects working on zkEVMs include:

- **Applied ZKP** - *Applied ZKP is a project funded by the Ethereum Foundation to develop an EVM-compatible ZK-rollup and a mechanism for generating validity proofs for Ethereum blocks.*
- **Polygon zkEVM** - *is a decentralized ZK Rollup on the Ethereum mainnet working on a zero-knowledge Ethereum Virtual Machine (zkEVM) that executes Ethereum transactions in a transparent way, including smart contracts with zero-knowledge-proof validations.*
- **Scroll** - *Scroll is a tech-driven company working on building a native zkEVM Layer 2 Solution for Ethereum.*
- **Taiko** - *Taiko is a decentralized, Ethereum-equivalent ZK-rollup (a Type 1 ZK-EVM).*
- **ZKSync** - *ZkSync Era is an EVM-compatible ZK Rollup built by Matter Labs, powered by its own zkEVM.*
- **Starknet** - *StarkNet is an EVM-compatible layer 2 scaling solution built by StarkWare.*

Further reading on ZK-rollups reading {#further-reading-on-zk-rollups}

- What Are Zero-Knowledge Rollups?
- What are zero-knowledge rollups?
- STARKs vs SNARKs
- What is a zkEVM?
- Intro to zkEVM
- Awesome-zkEVM resources
- ZK-SNARKS under the hood
- How are SNARKs possible?