How Zk-Rollups Work

Simon Brown

Follow

FCAT Blockchain Incubator

--

4

Listen

Share

Scaling Ethereum with Zero-Knowledge Proofs

Background

Scalability has always been an issue for public ledgers, this is old news. It's a problem that engineers have been trying solve for a number of years now. The issue has become more of a pressing need in the last 12 months with an increase of activity on public ledgers, and this has been reflected in transaction fees, sometimes seeing over $40 USD for common contract interactions on Ethereum.

The two main approaches to solving this issue have been a) scaling at the base layer, using techniques like sharding and novel consensus protocols, and b) layer 2 scaling techniques. Both approaches have been successful in different ways.

Layer 2 (L2)

Despite the progress that has been made on base layer scalability, it is clear that there will always be a place for layer 2 scaling techniques as well. This is for various reasons, from privacy, to building high performance, application specific networks, that leverage the security and liquidity of established public ledgers. Right now the main reason for adoption of L2 protocols is scalability. Layer 2 protocols can scale existing public ledgers very effectively, and offer a near to mid-term scaling solution for smart-contract platforms like Ethereum.

Existing L2 Scaling techniques

Channels

Channels are established between two parties, and allow them to send payments back and forth between each other instantly. These pair-wise, bi-directional channels can be meshed together to form a network of channels using P2P gossip protocols and probabilistic network routing. This is how the Lightning network works for Bitcoin. The same process can be used to do more than just payments, whereby generalized state channels can be constructed to allow for smart contract state transitions.

As with everything else, this approach has its trade-offs. Channels require an initial funding transaction, which results in a capital lockup for that channel. While this isn't really a big deal for a single channel, it does create complications when trying to route payments through a network, and the liquidity locked in intermediate channels makes it quite "capital inefficient". Channels also require both parties to actively monitor the base layer to ensure that the counterparty is adhering to the protocol.

Channels are the perfect solution for some use cases, and unidirectional payments are a great solution for service providers, to allow their customers to pay as they consume. Otherwise channels are best suited for a relationship between two entities that will be long-lived and will involve a great number of state updates.

Plasma

There were many flavours of Plasma that attempted to solve various issues in different ways. This plasma world map will give you an idea of the scope of the research, and the main types of Plasma constructions. However, every attempt at Plasma had some sort of trade-off that made for a poor user experience.

With Plasma Cash, tokens must have a fixed, non-divisible denomination, and users must keep a large amount of data for every token they have deposited. Plasma MVP involves complicated game theory to secure withdrawals, involving multiple

signatures per withdrawal transaction. Plasma Debit also involves very large sized proofs that the user must store, and it also requires a significant capital lockup by the operator in order to create pre-emptive channels.

All variations of Plasma are predicated on the assumption that the assets deposited have an explicit owner, which it turns out actually prevents creating more complicated applications such as AMMs or indeed an EVM.

Moreover, with Plasma sidechains, if something happens with the operator, there is no guarantee that another operator can take over and carry on the Plasma chain. This requires a trust in the operator. It also creates a “mass-exit” problem in the event that this does happen, whereby everyone in the Plasma chain tries to withdraw at the same time, causing huge congestion and high transaction fees on the base layer, and resulting in many people inevitably having to wait quite some time for their funds.

Rollups

Rollups have been described as a “hybrid” layer 2, in that they move computation off-chain, but keep data on-chain, which is one of the major differences from plasma, and the key to solving the data availability problem.

Rollups are similar to plasma in that they involve submitting a state commitment, or “state root”, to a smart contract on the Layer 1 EVM. This state root is derived from the state of all deposits / accounts in the rollup, using some sort of cryptographic accumulator, typically a merkle tree. When the state root is updated, a batch of all the transactions for that update is posted with the state root.

This means that anyone can take the data, which is essentially a series of state deltas, and can recreate the state of the rollup, and thus create new batches.

This gives security to the users of the rollup, in that it reduces the counterparty risk of a single operator, or collusion between multiple operators. Quite simply, the coordinator that is responsible for batching all the transactions in the rollup and posting them on-chain, could literally disappear in a puff of magical smoke, never to be seen again, and someone else can easily step in and pick up where they left off.

Some designs for rollups require multiple coordinators. For example, there is one rollup implementation that holds an auction for each batch to be published. Anybody can bid to become a coordinator, and if their bids succeeds, then they earn the right to create a batch of transactions, and keep the transaction fees from that batch.

Flavours of Rollups

The main difference between the two main types of rollups are in how the state transition is verified on-chain. Optimistic rollups rely on fraud proofs, while zk-rollups rely on zero-knowledge-proofs to verify the state transition.

Optimistic Rollups

An optimistic rollup can be described as an “honour system” to a certain extent, in that the smart contract on Layer 1 does not check the state transitions have been performed correctly. When a coordinator posts a new state root, the smart contract simply takes the coordinator at their word, and accepts the state update as-is.

It is only if somebody posts a “fraud proof” to the smart contract that the smart contract checks the state transition. It does this by verifying a fraud proof, which at a high level, consists of:

- a proof of “pre-state”, of how things looked before a transaction was applied
- a proof of “post-state”, which is how the operator claims the state should have looked after the transaction was applied
- a proof of the transaction that was applied during a state transition.

The rollup verifies all the proofs in order to verify that the pre-state, post-state and transaction were all included as part of a state transition submitted by a coordinator. The contract then applies the transaction logic to the pre-state, and compares the result to the post state. If there is a mismatch, then this proves that the coordinator did not apply the transaction properly. When this happens, the smart contract rolls back the state transition, and all subsequent state transitions until the latest one, and reverts to the last known valid state. The contract also slashes the coordinator’s stake, disincentivizing fraud or mistakes by coordinators and reducing the need for fraud proofs in the long run.

It turns out that this approach is very powerful. In fact, it allows for building some quite complex rollup constructions, to the extent that we can build a whole Layer 2 EVM in an optimistic rollup. This means that your Layer 1 contracts can be ported over to Layer 2 and still work more or less exactly the same.

The only real trade-off on this approach is that there needs to be a time-lock on withdrawals to allow for challenge periods. This can make large scale rollups quite “capital inefficient” and usually means involving liquidity providers to speed up

withdrawals.

It's worth noting that this description of optimistic rollups is somewhat of an over-simplification. In fact, if we were to "black-box" the zero-knowledge-proof mechanism in a zk-rollup, then the entire construction becomes less complex than optimistic rollups. For this reason, this post will focus on how zk-rollups work in theory, leaving a closer look at optimistic rollups and how they work, to a future post.

ZK-Rollups

Zk-Rollups were proposed in late 2018 / early 2019 by Barry Whitehat [1]. There are a number of implementations of zk-rollups in production, and they each implement the basic idea in different ways. I'm going to explain the basic principles based on Barry Whitehat's original design.

How It Works

Zk-rollups, like optimistic rollups, store all addresses and respective balances in a merkle tree. The root of this "balance tree" is stored in an on-chain smart contract.

When a batch is posted on-chain, a new merkle root is proposed, which reflects the updated merkle tree, with updated balances from all the transfers that are included in the batch. The smart contract will verify a snark proof that's posted with the batch, and if it checks out, the new merkle root is accepted, and becomes the canonical state root of the contract.

Deposits

When a deposit is made to a rollup, tokens are sent to the rollup contract. The rollup contract adds the tokens to a pending deposit queue. At a certain point, a coordinator will take a number of deposits and add them to the rollup, which basically involves including them in the merkle tree of balances.

How the deposits are actually added to the balances tree is a clever trick, that uses a technique called merkle mountain ranges [2].

Walkthrough

When a deposit is made to the smart contract, the smart contract takes all the relevant information (public key, amount, token type etc.) and creates a hash. This hash is added to the end of the deposit queue. Next the contract recursively hashes the last two deposit hashes in the queue, iteratively building up a merkle tree of deposit hashes. This merkle tree of deposit hashes becomes a sub-tree of the rollup balances tree. Let's walk through an example:

A deposit is made to the rollup contract so that the deposit queue now has one deposit:

Another subsequent deposit is made at a later stage:

The deposit queue now looks like this:

But now that we have an even number of deposits, the second deposit transaction continues the process by hashing the last two elements of the queue:

Now the deposit queue looks like this:

This single hash represents the two deposits that have been made to the smart contract, but how does the coordinator know that this is what the hash represents? How do we know that this single hash represents one deposit, two, or even sixteen? There are two ways that we know this:

1. each deposit emits an event which the coordinator can subscribe to, thereby being able to keep track of the details of each deposit made
2. the contract maintains a count of the size of the pending deposit queue and of the pending deposits sub tree height.

Let continue with our example:

Hopefully at this stage you can start to see a pattern emerge. We are following a simple recursive algorithm which continually builds up a merkle tree of deposits as they are added to the deposits queue. But how do these deposits move from the queue to the rollup?

Processing Deposits

The first thing to understand before we can go into how deposits are processed, is that all deposits and balances in the rollup are stored in a “sparse merkle tree”. This is essentially a merkle tree of a fixed size that is pre-initialized with all zeros, (or the equivalent hashes).

To process the pending deposits, a coordinator will take the first element of the pending deposits queue, and will insert it into the rollup’s sparse balance tree at the correct height. This will of course result in a new merkle root for the balance tree, which the coordinator posts to the rollup contract.

In order to accept this new balance tree merkle root, the contract must be satisfied that the coordinator is inserting the deposit subtree into a previously empty part of the balance tree. To satisfy the smart contract of this, the coordinator also posts a merkle proof of an empty node at the corresponding level of the balance tree, that will be replaced by the root of the deposit subtree. It looks a bit like this:

Recall that at this point the contract has everything it needs to verify that deposit subtree has been incorporated into the rollup correctly, namely:

- It has the sub tree height, which in this example is 2.
- It knows what an empty node at a height of 2 is, because it has a cache of empty node hashes for each level of the tree.
- It has just been given a merkle proof for this empty node so that it can verify that it is part of the current balance tree.
- It has the root of the deposit subtree (the first element in the deposit queue).

The contract now simply takes the empty node hash from the cache of node hashes, uses this hash with the given merkle proof to verify against the current balance tree root. If that checks out, it takes the hash from the start of the deposit queue, and then uses it in combination with the supplied merkle proof to derive the next balance tree root, and if this is the same as the balance tree root that the coordinator has just proposed, then the contract accepts it.

At this stage, a coordinator can give a merkle proof to any of the depositors, which they can use to independently verify that their deposit has been processed into the rollup.

If you have a keen eye, you’ll notice that when the coordinator processed the pending deposits from the queue, it only processed four of the deposits into the rollup, leaving one behind. This is because the coordinator can only take the “perfect tallest subtree” at any given time, which has to be a number that is a power of 2. The deposit(s) that are left behind need to wait for further deposits so that that number of deposits are a power of 2. In this case, even one other deposit would be enough, as then there would be 21 deposits at the top of the queue.

Transfers

Once clients have deposited their funds into the rollup, they are free to start transferring their funds to any other account in the rollup, quickly and cheaply. They do this by sending a transfer transaction to a coordinator, who then includes it in a batch with a load of other transactions, and submits it to the rollup contract, along with an updated balance tree root, and a zero knowledge proof.

The updated balance tree root is derived by simply updating all the balances in the balance tree according to the transactions in the batch. If Alice submits a transaction to say “send 10 tokens to Bob”, the coordinator will increase Bob’s balance by 10, reduce Alice’s balance by 10, re-hash the account data to get new account leaves, and rebuild the merkle tree. This results in a new merkle root, which gets sent to the smart contract. Easy.

Zero Knowledge Proof

This is where it gets interesting. Obviously the smart contract won’t take the coordinator at their word, the system is supposed to be trustless after all (and not optimistic). Therefore, the coordinator will create a proof that all the transactions were carried out correctly; let’s take a look at how this is done.

Recall that each coordinator maintains a local database of deposits, that builds up from subscribing to the deposit events emitted by the smart contract. When a transaction is received, the coordinator verifies it in the normal way by checking against the details in the database, it verifies:

- The transaction is correct and corresponds to the depositor’s public key.
- The depositor’s account exists in the balance tree.
- The amount being transferred is not greater than the depositor’s balance.
- The receiver account exists in the balance tree.

- Both the sender account and receiver account are of the same token type.
- The sender's nonce is correct.
- There are no overflows / underflows etc.

Once it checks out, it's added to the queue. Once a number of transactions are in the queue, the coordinator will decide to create a batch. To do this, the coordinator compiles a bunch of inputs for the zk-proof circuit to compile into a proof. This involves:

- Creating a merkle tree of all the transactions in the batch, which is padded with dummy transactions up to the size required by the circuit, and each with a merkle proof.
- Creating a collection of merkle proofs for each sender and receiver from all transactions in order to prove account existence.
- Creating a collection of intermediate roots which are derived from updating the balance tree root after updating the sender and receiver accounts in each transaction.

This last part is what allows the smart contract to be satisfied that the transactions in the batch were applied correctly. Let me explain.

The circuit has three public signals, these are operands that can be verified as being part of the proof, and they are the pre-state root, post-state root, and transaction root.

The circuit produces a proof by iterating over all the transactions, and performs all the same checks and verifications that the coordinator performed before accepting the transaction into the queue. On each iteration the circuit first checks that sender account exists under the current root using the account's merkle proof. Then it reduces the sender's balance and increments their nonce, it rehashes this updated account data, and uses this hash with the account's merkle proof to derive a new merkle root.

At this point, we know that this merkle root reflects the fact that the only thing that has changed

in the balance tree was the sender's balance and nonce. Why do we know this? Because we used the same merkle proof that proved the account's existence under the current balance tree root, to derive the new balance root.

Next, the circuit does exactly the same thing again, except with the receiver's account. That is: it verifies that the account exists under the new intermediate root, using the supplied merkle proof, it increases the balance accordingly, re-hashes the account data, and uses the same merkle proof to calculate a new balance tree root.

This entire process is repeated for each transaction, each iteration resulting in an updated state root from updating the sender's account, and a subsequent updated root from updating the receiver's account.

Each updated state root is a reflection of only one thing changing at a time. In this way, the circuit "walks" through the batch of transactions, creating a chain of updates, which results in a final balance tree root after the last transaction has been processed. This last final root becomes the new state root of the rollup.

L1 Finality

Once the circuit has completed creating a zero knowledge proof of all the state updates, the coordinator submits this proof to the smart contract. The circuit verifies the proof, and also examines the three public signals that form part of the proof. If you recall, those are:

- the pre-state root, i.e. the balance tree root as it is currently is before the batch is accepted
- the post-state root, i.e. the final root after applying all the state updates in the circuit
- the transaction tree root.

If the pre-state root in the contract matches the current balance tree as recorded in the contract, (and the proof is valid), then the contract takes the post-state root from the proof, and updates its current balance tree to match it.

At this point, any depositor can ask the contract to verify their balance by hashing their account data and supplying the contract with this hash, and a merkle proof to verify against the current account tree root.

Withdrawals

So what if a depositor wishes to withdraw their funds back to L1? It turns out that's fairly straightforward. Recall that one of the three public signals in our proof is the transaction root. This is the root of the merkle tree comprised of all the

transactions that were inputted into the circuit. Each transaction has an associated merkle proof that can be verified against this transaction root. When a batch is submitted to the contract, the contract records this transaction root, which allows any depositor to verify that their transaction was included in a batch, they simply supply the smart contract with the transaction details, a merkle proof and transaction root. The smart contract hashes the transaction data, verifies that it has a record of the transaction root, and verifies the transaction hash against this root using the supplied merkle proof.

In order to make a withdrawal, a depositor sends funds to the account at index 0 in the balance tree. The account at this index is reserved for this purpose, and sending funds to this address burns those funds on L2.Once the transaction has been included in a batch, the depositor sends a withdrawal request to the smart contract, with proof of their transaction to the "burn account", using the mechanism described above.

The withdrawal request contains the transaction details, merkle proof, transaction tree root and an L1 address to transfer the tokens to. Once the transaction existence has been verified, the smart contract checks that the withdrawal hasn't already been processed, and then sends the funds the specified recipient at L1.

Atomic Swaps

I should perhaps call them "dependent transactions

", which is probably a better description. So far our zk-rollup has allowed us to deposit to an L2 rollup, transfer funds to other recipients on L2, and withdraw back to L1. But this gives us fairly limited capability, it really only gives us a payment network. If we introduce dependent transactions, we can start building some interesting things at the layer above. For example, we could build an order book exchange, matching orders and then including each reciprocal transaction pair as being dependent on each other.

This is as simple as including the transaction hash of the counterparty transaction in the transaction in each reciprocal pair. (Obviously the counterparty transaction hash cannot be part of the hash itself, or this will create a circular dependency, but there are ways of getting around this).

Data Availability

Recall that one of the principal differences between rollups and plasma, was that rollups are hybrid L2 protocols. That means that they take computation off the base layer, but they keep the data on layer 1

. This is important, as it means that anybody can take the data that is posted to the base layer, and use it to re-build the rollup and start accepting transactions and creating batches and so on.

In order to make it feasible to use the base layer as a data availability layer, the transactions are compressed and posted to the smart contract as calldata. This saves a lot of space, and at 16 gas per byte, saving space means saving gas. And that is where we are able to achieve a high transaction throughput.

According to the Solidity docs: "Calldata is a non-modifiable, non-persistent area where function arguments are stored, and behaves mostly like memory." [3]

The reason for using calldata is simply that it is the cheapest form of storage to use. In fact, the state-deltas passed as a calldata parameter doesn't get stored in Ethereum's state at all, but Ethereum nodes can store the transaction data as the block is created.

So what do these state deltas look like and how much can we save through data compression? As you can see from the below table, there are several adjustments we can make to how we store transaction data with rollups, that result in significant savings in data storage:

All the metrics above are measured in bytes. When we visualize these numbers side-by-side, the savings we get are quite striking:

Benchmarking

So how does saving space in terms of the data posted on-chain actually translate into a high transaction throughput? Well the answer to this question is has to do with the gas costs, and more importantly, Ethereum's block gas limit.

Basically, there is an upper bound to the amount of gas that can be processed in a block, and this has to be shared with all the transactions in that block. That means there is a relative upper limit to the amount of gas a transaction can consume. That means that the less space each individual state-delta occupies, the more of them you can fit in a single transaction. Let's break it down.

If we divide the block gas limit by the average size of a zk-rollup block containing 2048 transactions, we can see that we can fit up to

23 batches per block.

A quick calculation to figure out our transactions per second we get:

This is obviously only a theoretical upper bound on the number of batches we can publish in a single layer 1 block, because of course we will need to share each block with other non-rollup transactions. That being said, it gives us an indication of roughly how far we can push our transaction throughput. If half the transactions in a block were some sort of Layer 2 rollup, we're looking at 1,500 TPS.

You might be wondering why I chose 2048 transactions per batch for my calculations? Well ~2,000 transactions seem to be what existing zk-rollup solutions are boasting at the moment. Creating a zero-knowledge proof that processes thousands of accounts and deposits is very computationally expensive, and there are limits with today's technology.

Future Research

An area that would be worth researching is how expensive it is to create a proof. I'd like to see a benchmark that maps metrics such as number of transactions per batch, number of accounts in rollup, and how long it takes to create a proof given a certain amount of processing power and memory. Of course any such benchmarking would have to take into account the particular proving system used (groth16, plonk, stark etc.).

Another interesting area of exploration would be to look at which other EVM platforms support zk-rollups. Ethereum introduced several optimizations, and gas price restructuring, in the Istanbul upgrade, which were specifically aimed at optimizing smart contract execution to support zk-rollups. In particular, EIP-1108 which reduced the cost of using pre-compiled contracts for elliptic curve operations on the BN128 curve, and EIP-2028, which reduced the cost for calldata parameters to 16 gas per byte. It would be interesting to see if other EVM chains support snark verification in the same way, and whether you can build a rollup for these chains.

Interoperability

Certain ideas have been proposed to allow for cross-rollup communication. One idea for cross-L2 transactions that doesn't require a withdrawal-and-deposit approach via L1 is something called mass migrations [4]. This basically involves batch transfers to another rollup, which would be processed by the L1 smart contract, and subsequently merged into the state tree of the destination rollup using a similar process to normal deposits. There is some extra logic around the validation of the mass migration, and of course, there needs to be standardization between rollups in order for this to occur, but so far this seems to be a very promising direction.

Conclusion

At the moment, zk-rollups offer a very practical solution for scaling Ethereum. High transaction throughput with very low cost transactions and instant finality on Layer 2, and roughly ten minute finality on Layer 1, makes zk-rollups a very appealing scaling solution.

One current drawback of the zk-rollup approach is the lack of smart contract support on layer 2. However, with several zk-rollup implementations that support smart contracts (or even EVMs) in late stages of development, it would seem this drawback will soon be a thing of the past.

One question still to be answered is how difficult it will be to run a coordinator node, and the effects this has on decentralization. For example, does it require very expensive hardware in order to be able to generate proofs?

To a certain a degree it really doesn't matter that much because zk-rollups leverage the security guarantees of the base layer, and are mostly trust-minimized, non-custodial protocols. However, having a number of coordinators gives even stronger security guarantees and is definitely an attractive selling point for the platforms that support it.

Another question to ask is whether the emergence of highly scalable layer 1 protocols. will render the use of rollups redundant in a few years. To a certain extent it is probable that there will always be institutions and organizations who will make use of rollup style techniques for a variety of reasons. These reasons could include building highly optimized application specific rollups, to creating interoperable, trust-minimized private networks that leverage the standards and security of established public ledgers.

Thanks for reading this deep dive on zk-rollups. If you've spotted any inaccuracies, please do me a favour let me know, I'd appreciate it.

eReview #986478.1.0