CONTENT_DISCLAIMER.md: Polygon Knowledge Layer third-party content disclaimer The Polygon Knowledge Layer contains third-party content which includes websites, products, and services that are provided for informational purposes only. Polygon Labs does not endorse, warrant, or make any representations regarding the accuracy, quality, reliability, or legality of any third-party websites, products, or services. If you decide to access any third-party content linked from the Polygon Knowledge Layer, you do so entirely at your own risk and subject to the terms and conditions of use for such websites. Polygon Labs reserves the right to withdraw such references and links without notice. The Polygon Knowledge Layer serves as an industry public good and is made available under the MIT open source license. In addition, please view the official Polygon Labs Terms of Use. # README.md: Polygon Knowledge Layer Welcome to the Polygon Knowledge Layer. These docs use the Material theme for MkDocs. Our goal is to establish a high-quality, curated, and comprehensive "source of truth" for Polygon's technology. This includes sections on: - Polygon CDK - Polygon zkEVM - Polygon PoS - Polygon Miden - Developer tools Run locally Prerequisites 1. Python 3.12. 2. virtualenv: Install using pip3 install virtualenv. Setup 1. Clone the repository. 2. cd to the root. 3. Run the run.sh script. You may need to make the script executable: chmod +x run.sh sh ./run.sh The site comes up at http://127.0.0.1:8000/ Docker If you prefer Docker, you can build and run the site using the following commands: sh docker build -t polygon-docs . docker compose up Contributing Getting started 1. Fork and branch: Fork the main branch into your own GitHub account. Create a feature branch for your changes. 2. Make changes: Implement your changes or additions in your feature branch. 3. Contribution quality: Ensure that your contributions are: - Atomic: Small, self-contained, logical updates are preferred. - Well documented: Use clear commit messages. Explain your changes in the pull request description. - Tested: Verify your changes do not break existing functionality. - Styled correctly: Follow the Microsoft Style Guide. Creating a pull request 1. Pull request: Once your changes are complete, create a pull request against the main branch of Polygon Knowledge Layer. 2. Review process: Your pull request will be reviewed by the maintainers. They may request changes or clarifications. 3. Responsibility: Contributors are expected to maintain their contributions over time and update them as necessary to ensure continued accuracy and relevance. Best practices - Stay informed: Keep up-to-date with the latest developments in Polygon technologies. - Engage with the community: Participate in discussions and provide feedback on other contributions. - Stay consistent: Ensure your contributions are coherent with the rest of the documentation and do not overlap or contradict existing content. Contact and support - For docs issues (technical or language) open an issue here. - For technical issues with the software, either raise an issue here and we will follow up, or check https://support.polygon.technology/support/home. The team - Anthony Matlala (@EmpieichO) - Hans Bodani (@hsutaiyu) - Katharine Murphy (@kmurphypolygon) # index.md: --- hide: - navigation - toc ---

The Polygon Knowledge Layer

Welcome to the technical documentation and knowledge resources for Polygon protocols and scaling technologies.

Learn how to build and deploy dApps, launch ZK rollups and validiums as Layer 2s on Ethereum, spin up nodes, and find out about the latest in zero-knowledge research.

BUILD

Build today using Polygon technology. Select the protocol that best fits your needs.

[Polygon PoS
LIVE]

[Deploy a dApp on the widely adopted Polygon Proof-of-Stake protocol, an EVM-compatible environment optimized for high throughput and low transaction fees].

[Polygon zkEVM
LIVE]

[Deploy a dApp or build infrastructure on zkEVM, an EVM-equivalent ZK rollup designed for security].

[Polygon CDK
LIVE]

[Build and test a zero-knowledge Layer 2 blockchain on Ethereum. Learn about validium and rollup modes, custom native gas tokens, and more].

[Polygon Miden
COMING SOON]

[Test the Miden VM and learn about Polygon Miden, the novel ZK rollup designed to extend the EVMâ€™s feature-set, including for privacy].

[Polygon Edge
DEPRECATING]

[Polygon will shortly be removing support for Edge. The documentation is now managed in the Edge repo].

LEARN

Deep dives only. Further your understanding of Polygon scaling technology.

[AggLayer]

[Introducing the multi-chain, multi-transaction, Polygon AggLayer; what it is and how it works].

[Innovation & design]

[Resources focused on both current and future Polygon technologies. It features detailed guides, foundational concepts, and previews of upcoming innovations.]

[Type 1 prover]

[The Polygon type 1 proving component used for creating proofs on your ZK-EVM chain].

[Plonky 2 & 3]

[Keep up with our latest cryptographic developments with the Plonky 2 & 3 libraries].

[Polygon protocols]

[The Polygon protocol thatâ€™s best for you. A guide and decision matrix designed to empower users to navigate the evolving world of decentralization].

Developer resources

For developers who know what they want to build and are ready to go.

Developer tools

[RPC providers, faucets, data indexing, Web3 dApp development SDKs, block explorers, storage, and more].

[Write a zkEVM contract]

[Step-by-step guidance for writing smart contracts with zkEVM].

[Solution Provider Network]

[Searchable catalog of tooling and infrastructure for developers.]

Quickstart

Are you ready to start building?

[Polygon CDK: Deploy a local test rollup]
[Polygon zkEVM: Deploy a smart contract to the zkEVM Cardona testnet]
[Polygon PoS: Build a new web3 dApp]
[Polygon PoS: Bridge tokens and send interlayer messages]
[Polygon zkEVM: Set up a zkNode]
[Polygon CDK: Create your own validium]
[Polygon Miden: Explore the sandbox]

agglayer-go.md: The AggLayer-go is a service designed to receive zero-knowledge (ZK) proofs from various CDK chains and verify their validity before sending them to the L1 for final settlement. !!! warning This service is being deprecated in favor of the more robust and efficient Rust implementation. Architecture The AggLayer Golang architecture supports interactions with multiple CDK chains for proof-verification. It uses a PostgreSQL database for storage and interacts with both L1 and L2 chains through configured RPC nodes. The diagram below shows the full start-up, running, and shutdown sequence for the application and its components.

!CDK architecture

Get started Run locally with Docker 1. Clone the repository: bash git clone https://github.com/AggLayer/agglayer-go 2. Execute the following command: bash make run-docker Production set up 1. Ensure only one instance of the AggLayer is running at a time. 2. Use a containerized setup, or OS level service manager/monitoring system, for automatic restarts in the case of failures. Prerequisites Hardware - For each CDK chain it's necessary to configure its corresponding RPC node, synced with the target CDK. - This node is for checking the state root after executions of L2 batches. - We recommend a durable HA PostgresDB for storage, preferably AWS Aurora PostgreSQL or Cloud SQL for PostgreSQL in GCP. Software - Docker - Docker Compose Installation 1. Clone the repository: bash git clone https://github.com/AggLayer/agglayer-go 2. Install Golang dependencies: bash go install . Configure key signing 1. Install polygon-cli: bash go install github.com/maticnetwork/polygon-cli@latest 2. Create a new signature: bash polygon-cli signer create --kms GCP --gcp-project-id gcp-project --key-id mykey-tmp 3. Install gcloud CLI and set up ADC: bash gcloud auth application-default login 4. Configure KMSKeyName in agglayer.toml. Configure agglayer.toml [FullNodeRPCs] to point to the corresponding L2 full node. [L1] to point to the corresponding L1 chain. [DB] section with the managed database details. API Refer to the cmd and client directories for API implementation details. Documentation and specific API routes can be generated from these sources. --- For more information, visit the AggLayer-go repository. # agglayer-rs.md: AggLayer-rs is a Rust-based service designed to receive ZK proofs from various CDK chains and verify their validity before sending them to the L1 for final settlement. It replaces the previous Golang implementation. Architecture The AggLayer Rust architecture supports interactions with multiple CDK chains for proof-verification. Its architecture is the same as agglayer-go, but without the PostgreSQL database for storage. Getting started Prerequisites Hardware - RPC nodes: Configure RPC nodes for each CDK chain, and synced with the target CDK, to check the state roots post L2 batch executions. Software - Rust Installation 1. Clone the repository: sh git clone https://github.com/AggLayer/agglayer-rs.git cd agglayer-rs 2. Build and run: sh cargo build cargo run How to Configure the service Edit configuration by modifying the agglayer.toml file for service settings like RPC endpoints and network parameters. For example: toml [network] rpcurl = "https://yourrpcurl" chainid = "yourchainid" Run tests 1. Run unit tests: sh cargo test 2. Run integration tests by first ensuring all necessary services are running, then execute: sh cargo test -- --ignored API reference Endpoints Submit proof - Endpoint: /submitzkp - Method: POST - Payload: json { "zkp": "base64encodedzkp", "chainid": "cdkchainid" } - Response: json { "status": "success", "message": "ZKP submitted successfully" } --- For more information, visit the AggLayer Rust repository. # overview.md: !!! info "Disclaimer" - Some of the content in this section discusses technology in development and not ready for release. - Please check against the main documentation site for any live releases. - Feel free to experiment with any code in public repos. The AggLayer is an in-development interoperability protocol that allows for trustless, cross-chain token transfers and message-passing, as well as more complex operations. The safety of the AggLayer is provided by ZK proofs. The AggLayer currently connects chains built with Polygon CDK, a developer toolkit for designing ZK-powered Layer 2s. The long term goal for the protocol is to be flexible enough to provide interoperability among a growing range of blockchain architectures, including L2s, appchains, and non-EVM chains. Chains connected to AggLayer Here's a list of chains connected to the alpha version of the AggLayer: | Implementation Provider | Chain Name | L2 Chain ID |Network Name | | ------------------ | ------------- | ----------- | ----------------------- | | Startale Labs | Astar | 6038361 | zKyoto | | Startale Labs | Astar | 3776 | Astar zkEVM | | Gateway FM | GPT Protocol | 1511670449 | gpt-mainnet | | Gateway FM | Haust | 1570754601 | haust-testnet | | Gateway FM | Haust | 938 | haust-network | | Gateway FM | Lumia | 1952959480 | Lumia Testnet | | Gateway FM | Lumia | 994873017 | prism | | Gateway FM | Moonveil | 1297206718 | moonveil-testnet | | OKX | OKX | 196 | X Layer | | Gateway FM | Silicon | 1722641160 | silicon-sepolia-testnet | | Gateway FM | Silicon | 2355 | silicon-zk | | Gateway FM | WilderWorld | 1668201165 | zchain-testnet | | Gateway FM | Wirex | 31415 | pay-chain | | Gateway FM | Witness | 1702448187 | witnesschain | AggLayer components Polygon CDK The AggLayer connects chains built with Polygon CDK, which use ZK proofs to generate state transitions that are cryptographically secure. Unified bridge The unified bridge is a single bridge contract for all AggLayer-connected chains, allowing for the cross-chain transfer of fungible (non-wrapped) tokens. It is the source of unified liquidity for the AggLayer. !!! tip "More information" See the unified bridge documentation for details. AggLayer service The AggLayer service is designed to receive ZK proofs from various CDK and non-CDK chains, and verify their validity before sending them to the L1 Ethereum for final settlement. Currently, the AggLayer service has two implementations: agglayer-go and agglayer-rs. # token-flows.md: The following workflows describe how token transfers and message passing are implemented by the unified bridge across various L1 and L2 permutations. While the descriptions refer only to token transfers in the current AggLayer implementation, the sequence of events is exactly the same for arbitrary messages. L1 to L2 1. If a call to bridgeAsset or bridgeMessage on L1 passes validation, the unified bridge contract appends an exit leaf to the L1 exit tree and computes the new L1 exit tree root. 2. The global exit root manager appends the new L1 exit tree root to the global exit tree and computes the global exit root. 3. The sequencer fetches the latest global exit root from the global exit root manager. 4. At the start of the transaction batch, the sequencer stores the global exit root in special storage slots of the L2 global exit root manager smart contract, allowing L2 users to access it. 5. A call to claimAsset or claimMessage provides a Merkle proof that validates the correct exit leaf in the global exit root. 6. The unified bridge contract validates the caller's Merkle proof against the global exit root. If the proof is valid, the bridging process succeeds; otherwise, the transaction fails. L2 to L1 1. If a call to bridgeAsset or bridgeMessage on L2 passes validation, the unified bridge contract appends an exit leaf to the L2 exit tree and computes the new L2 exit tree root. 2. The L2 global exit root manager appends the new L2 exit tree root to the global exit tree and computes the global exit root. At that point, the caller's bridge transaction is included in one of the batches selected and sequenced by the sequencer. 3. The aggregator generates a zk-proof attesting to the computational integrity in the execution of sequenced batches which include the transaction. 4. For verification purposes, the aggregator sends the zk-proof together with all relevant batch information that led to the new L2 exit tree root (computed in step 2), to the verifier contract. 5. The verifier contract utilizes the verifyBatches function to verify validity of the received zk-proof. If valid, the contract sends the new L2 exit tree root to the global exit root manager in order to update the global exit tree. 6. claimMessage or claimAsset is then called on the unified bridge contract with Merkle proofs for correct validation of exit leaves. 7. The unified bridge contract retrieves the global exit root from the L1 global exit root manager and verifies validity of the Merkle proof. If the Merkle proof is valid, the settlement completes, otherwise, the transaction is reverted. L2 to L2 1. When a batch of transactions is processed, the bridge contract appends the L2 exit tree with a new leaf containing the batch information. This updates the L2 exit tree root. 2. The bridge contract communicates the L2 exit tree root to the L2 global exit root manager. The L2 global exit root manager, however, does not update the global exit tree at this stage. 3. For proving and verification, the zk-proof-generating circuit obtains the L2 exit tree root from the L2 global exit root manager. 4. Only after the batch has been successfully proved and verified does the L2 global exit root manager append the L2 exit tree root to the global exit tree. As a result, the global exit root is updated. 5. The zk-proof-generating circuit also writes the L2 exit tree root to the mainnet. The L1 bridge contract can then finalize the transfer by using the claim function. # unified-bridge.md: The unified bridge is a single bridge contract on Ethereum, providing a safe, common access point for the transfer of all native, never-wrapped tokens. Each chain has a local copy of the unified bridge root, enabling cross-chain interoperability that doesnâ€™t require the security risks of third-party bridges. !!! important "AggLayer smart contracts" - The current version of the unified bridge uses the contracts in the PolygonzkEVM smart contract repo. Bridging mechanism The bridging mechanism enables token transfers and message-passing between Ethereum (L1) and CDK chains via smart contracts. Detailed in the zkEVM bridging documentation, core components include the bridge and exit root Solidity smart contracts. Data structures Each chain holds a single data structure which stores a record of all token withdrawals and messages that originated from that chain. This â€œexit treeâ€ is an append-only Merkle trie, similar in structure to the Ethereum deposit trie. The latest state of each chain and the unified bridge is represented by the root of this Merkle tree, referred to as the â€œexit root.â€ As cryptographic commitments, exit roots ensure the integrity of the network as a whole. Refer to the exit root documentation for greater detail about the role of exit roots in the system. # index.md: --- hide: - toc ---

Polygon CDK

Polygon Chain Development Kit (CDK) is a modular, open-source blockchain stack for developers launching sovereign L2 chains powered by zero-knowledge (ZK) proofs.

Understand the CDK

A high-level overview of the CDK, its features and benefits.

Try out the CDK locally

Get started by deploying a chain on your local machine.

CDK concepts

Learn about the concepts behind the CDK.

CDK architecture

Dive deeper into the components of a CDK chain and how they interact.

Contribute to the CDK

Get involved in building the open-source CDK stack on GitHub.

Join the dev community

Join our developer Discord server to ask questions and get help.

overview.md: The Polygon Chain Development Kit (CDK) is a modular, open-source software toolkit that enables blockchain developers to deploy and configure ZK-powered chain architectures. With Polygon CDK, developers can launch new chains that use the Polygon zkEVM protocol as custom Layer 2 solutions on Ethereum. Developers can configure ZK-powered rollups, validium networks, or even sovereign chains with their own tailored finality mechanisms. Trustless finality Why does Polygon CDK focus on ZK-powered chains? Polygon CDK focuses on ZK-powered chains because zero-knowledge technology enables chains to achieve trustless finality, which means chain users don't need to rely on a small group of individuals to confirm the finality of their transactions, ensuring greater security and a higher degree of decentralization. With Polygon CDK, developers can choose between a rollup and validium chain setup, and decide on how they want to configure the chain's proving system. Polygon has launched the CDK rollup/validium mode enabling developers to configure CDK components that run the Polygon zkEVM protocol. Key advantages Polygon CDK provides the essential components to build a Layer 2 blockchain that is secure, scalable, and interoperable with other chains. - Security: CDK builds highly secure, scalable L2s that utilize the latest innovations in zero-knowledge technology. - High-performance: CDK Erigon implements fast-syncing Erigon clients that run the battle-tested Polygon zkEVM protocol. - Modularity: CDK modular components allow developers to easily customize their L2 environment and build a chain that meets their specific needs. - Interoperability: Opt-in to the AggLayer to bootstrap your chainâ€™s ecosystem, enable cross-chain transactions while expanding your reach, user base, and liquidity from other established chains. - Sovereignty: Maintain full control over your chainâ€™s revenue, governance, security, economic policies, and more. - Low gas fees: Transaction fees are orders of magnitude lower than those on Ethereum and are processed substantially faster. This enables a fast, ultra low-cost, and secure user experience unaffected by any high activity experienced on shared networks. Next steps - Deploy a local CDK environment using Kurtosis. Follow the guide to deploy a CDK stack on your local machine. - Check out the concepts documentation to gain a high-level understanding of the CDK. - Have a look at the CDK architecture docs to understand the CDKâ€™s components and how they interact with each other. # cdk-validium.md: The Polygon CDK validium is one of two configuration options of the Polygon CDK, the other being the Polygon zkEVM rollup. As per the definition of validium, the Polygon CDK validium uses validity proofs to enforce integrity of state transitions, but it does not store transaction data on the Ethereum network. The Polygon CDK validium is in fact a zero-knowledge validium (zkValidium) because it utilises the Polygon zkEVM's off-chain prover to produce zero-knowledge proofs, which are published as validity proofs. The use of the above-mentioned prover, to a certain extent, adds trustlessness to the Polygon CDK validium. The validium mode inherits, not just the prover, but all the Polygon zkEVM's components and their functionalities, except that it does not publish transaction data on L1. The validium configuration has one major advantage over the zkEVM rollup option: And that is, reduced gas fees due to the off-chain storage of transaction data, where only a hash of the transaction data gets stored on the Ethereum network. Data availability committee (DAC) In relation to storing transaction data off-chain, the CDK validium comes with the requirement to manage the data. - First of all, the transaction data is not published to the L1 but only the hash of the data. - Secondly, a trusted-sequencer collects transactions from the transaction pool manager, puts them into batches and computes the hash of the transaction data. It is due to the above two points that the Polygon CDK validium has to have a set of trusted actors, who can monitor and even authenticate the hash values that the sequencer proposes to be published on the L1. The hash values need to be verified as true footprints of the transaction data corresponding to all transactions in the sequenced batches. These trusted actors are collectively called the Data Availability Committee (DAC). After verifying the proposed hash values individually, each DAC member signs them and sends the signature to the sequencer. The sequencer uses a multi-sig, which is a custom-specified m-out-of-n multi-party protocol, to attach the required m signatures to the hash of the transaction data. The multi-sig contract lives on the L1 network. Architecturally speaking, the Polygon CDK validium is therefore nothing but a zkEVM with a DAC. That is, Polygon CDK validium = Polygon zkEVM + DAC. Validium data flow The DAC works together with the sequencer to control the flow of data and state changes. The diagram below depicts a simplified outline of the Polygon CDK validium architecture. It particularly shows how the DAC and the sequencer relate in the overall data flow. !CDK validium data availability dataflow The entire process can be broken down as follows: 1. Batch formation: The sequencer collects user transactions, adds them to blocks, and puts the blocks in batches while recursively computing their hash values. 2. Batch authentication: Once the batches are assembled, and their hash values computed, they need to be authenticated by the DAC. The sequencer therefore forwards the batch data, and their corresponding hash values to the DAC, as a way to request for signatures. 3. Data validation and storage: The DAC nodes independently validate the batch data against the hash values received from the sequencer. Once validated, each hash value is stored in each DAC node's local database for future reference. 4. Signature generation: Each DAC node generates a signature for each batch hash. This serves as an endorsement of the batch's integrity and authenticity. 5. Communication with Ethereum: The sequencer collects the DAC members' signatures and the original batch hash, and submits them to the Ethereum network for verification. 6. Verification on Ethereum: A designated multi-sig smart contract on Ethereum verifies the submitted signatures against each DAC member's known signatures, and confirms that sufficient approval has been provided for the batch hash. 7. Final settlement with zero-knowledge proof: The aggregator prepares a proof for the batch via the prover and submits it to the Ethereum network. This proof confirms the validity of the transactions in the batch without revealing transaction details. The chain's state gets updated on Ethereum. # cdk-zkevm.md: Polygon zkEVM is a zero-knowledge rollup (or zk-rollup) designed to emulate the Ethereum Virtual Machine. It is a scaling-solution to Ethereum as it rolls up many transactions into one batch. These batches are submitted to the L1, where their integrity is proved and verified before being included in the L1 state. Proving, verification of batches, and state changes are all controlled by smart contracts. Most important to understand is the primary path taken by transactions from when users submit these transactions to the zkEVM network up until they are finalized and incorporated in the L1 State. Polygon zkEVM achieves this by utilizing several actors. The diagram below depicts the various actors and how they interact. !zkEVM option architecture Here is an outline of the most prominent rollup components: - The users, who connect to the zkEVM network by means of an RPC node (e.g., Infura or Alchemy), submit their transactions to a database called the pool DB which is managed by the transaction pool manager. - The pool DB is the storage for transactions submitted by users. The transaction pool manager selects the transactions to send to the sequencer. - The sequencer is a node responsible for receiving transactions, checking if the transactions are valid, then putting valid ones into a batch. The sequencer submits all batches to the L1 and then sequences the batches. This process proposes the sequence of batches to be included in the L1 state. - The state DB is a database for permanently storing state data (but not the Merkle trees). - The synchronizer is the component that updates the state DB by fetching data from Ethereum through the Etherman. - The Etherman is a low-level component that implements methods for all interactions with the L1 network and smart contracts. - The aggregator is another node whose role is to produce proofs attesting to the integrity of the sequencer's proposed state-change. These proofs are zero-knowledge proofs (or ZK-proofs) and the aggregator employs a cryptographic component called the prover for this purpose. - The prover is a complex cryptographic tool capable of producing ZK-proofs of hundreds of batches, and aggregating these into a single ZK-proof which is published as the validity proof. # high-level-views.md: CDK full execution proof (FEP) The diagram below depicts a simplified architectural layout of the CDK FEP stack and indicates at a high level how components communicate. !High level view of CDK stack Component interactions - The CLI tool is the starting point. Developers, or chain administrators in particular, can use the CLI tool to build and configure chains in various modes of operation, such as validiums and rollups. - Once a chain is configured with the CLI, users can submit transactions through the CDK Erigon RPC node. These transactions are relayed to the tx-pool manager before the sequencer selects and executes them. - The sequencer sequences transactions batches and synchronizes data with the RPC node. - The sequencer sender reads batch data from the RPC node. - The aggregator reads batch data from the sequencer data stream. - The sequencer sender persists data into the L1 smart contract domain for rollup mode and into DAC nodes for validium mode operations. - The aggregator sends batches to the prover and receives proofs in return. Together with the prover, it aggregates the proofs into batches before submitting the final proofs to the AggLayer or L1, depending on the chosen settlement layer. - Users interact with the bridge

service via the bridge UI or API. - The AggLayer verifies proofs and interacts with the L1 smart contracts. User data flow The following diagram is a sequential depiction of the user data flow for the CDK FEP stack in validium mode using a mock prover and having an AggLayer connection. !High level view of CDK user data flow Sequential interactions 1. User sends a transaction to the CDK Erigon RPC node. 2. The CDK Erigon RPC node proxies the data to the CDK Erigon sequencer node and syncs the batch data between the sequencer and the RPC nodes. 3. The sequencer sequences the transaction batches. 4. The sequencer sender reads batches from the RPC node. 5. In validium mode only, the sequencer sender persists transaction data into the DAC nodes. 6. The sequencer sender sequences the batches into the L1 smart contracts. 7. The aggregator reads batches from the sequencer data stream. 8. The aggregator sends batches to the provers. 9. The aggregator submits the final proof to the AggLayer. 10. The AggLayer submits the final proof to the L1 smart contract domain. mermaid sequenceDiagram participant User participant ErigonRPC as CDK Erigon RPC Node participant Sequencer as CDK Erigon Sequencer Node participant SeqSender as Sequence Sender participant Aggregator participant AggLayer participant DACNodes as DAC Nodes participant Prover participant L1 as L1 Smart Contracts User->>ErigonRPC: Send transaction ErigonRPC->>Sequencer: Proxy and sync transaction data Sequencer->>Sequencer: Sequence transaction batches SeqSender->>ErigonRPC: Reads batches SeqSender->>DACNodes: Persist transaction data (validium mode only) SeqSender->>L1: Sequence batches into L1 Smart Contracts Aggregator->>Prover: Send batches to Prover Prover->>Aggregator: Return proofs Aggregator->>Aggregator: Aggregate the proofs Aggregator->>AggLayer: Submit final proof AggLayer->>L1: Submit final proof to L1 Smart Contract Domain !!! tip Detailed AggLayer flows will be published soon. # staking-the-bridge.md: The LxLy bridge is the native bridging infrastructure for all CDK chains. The way it works is each individual CDK chain deploys an instance of the LxLy bridge that connects to an L1 (Ethereum by default) by deploying contracts that carry out deposit and withdrawal of assets, along with escrow management. These contracts are managed by node operators corresponding to the respective CDK chains. This changes as the AggLayer v1 goes online, and introduces an upgrade to the existing LxLy architecture in the form of a unified instance of the LxLy bridge that multiple chains can connect to. What's the "unified" bridge? AggLayer envisions a scalability solution that leverages shared state and unified liquidity across multiple ZK-powered chains within the Polygon ecosystem, all powered by the CDK infrastructure. !!! tip "What's AggLayer?" Want to learn more about what AggLayer is and what it looks to achieve? Check out the "WTF is Polygon" article in Polygon blog. All of this cool infrastructure needs a unified channel for easy transmission of assets and messages between the multiple chains connected via the AggLayer. And this is where the unified bridge comes into play. It allows all chains to take advantage of the AggLayer's unified liquidity, lower transaction costs, and more. !!! tip "Lxly vs unified bridging TL;DR" - LxLy bridge: A ZK bridge that supports asset and message transfers between a zkEVM system and the L1, typically Ethereum. - Unified bridge: A specific instance of an LxLy bridge that allows several chains to connect to it. This instance is specific to the AggLayer v1. The new unified model of the LxLy bridge introduced as a part of the AggLayer v1 infrastructure has one significant difference from the existing LxLy bridge: any asset bridged onto a CDK chain using the unified bridge is held by the Unified Escrow (also referred to as the Master Escrow) contract instead of a dedicated bridge contract. Due to the shared nature of the bridge, chain operators will not have admin access to the funds locked in the master escrow contract, including the funds that belong to their own network. The ability to manage bridge reserves is crucial to implement staking/restaking and other similar use cases, and is managed by the respective chain operators. How does the unified bridge address this? Introducing "Stake the Bridge" Stake the Bridge, or STB, is a feature that lets CDK chain operators maintain control over the assets that are deposited to their respective networks. Design and implementation On L1, CDK chains enable STB for an asset by deploying STB contracts on L1 to create an alternative L1Escrow account that holds the asset, and allows the CDK chain operator to manage this token reserve. On L2 (the CDK chain), there are three components needed to make this work: - L2Token, which is a natively deployed ERC20 contract. - L2Escrow, a contract that manages the L2Token's supply. - L2TokenConverter, the contract that enables converting bridge-wrapped tokens to natively-minted tokens on L2. !!! info Each token needs to have its own set of STB contracts that perform the functions described above. Let's briefly go over the specific actions and characteristics of each STB contract on L1 and L2. L1Escrow - Defines staking strategies to contribute to network security, or achieve other goals. - Sending token issuance messages to L2Escrow. - Fulfilling redemption messages from L2Escrow. L2Escrow - Receives minting instructions from L1Escrow via the unified bridge upon token deposit, and prompts L2Token contract to mint assets to a given address on L2. - Burns the native asset on L2 and sends minting instructions to L1Escrow to release assets on L1. L2Token - Natively mints L2 tokens and sends them to a designated address. - Interfaces with the L2TokenConverter contract. L2TokenConverter - Supports 1:1 conversion between the STB minted native tokens, and the bridged tokens minted by depositing tokens to LxLy bridge directly on L1. - Doesn't have a default cap on the token volume that can be converted, but it can be added by chain operators as necessary. - An asset can have multiple token converters that can have different properties. Roles and responsibilities There are three roles, each of which performs specific actions to manage the STB system: - Admin: Can upgrade and pause the system. - Escrow manager: Can withdraw token backing from the respective escrow contract to stake using the managerWithdraw() function and contribute to network security, or achieve other goals. - Risk manager: Can invoke setIssuanceCap() multiple times to increase or reduce the issuance cap. STB transaction flow vs. existing LxLy flow With the STB contracts set up on L1 and L2 for a particular CDK chain, the bridging UX for a user doesn't differ from what it would be if they were carrying it out using the existing LxLy bridge. Let's consider an example using the following tokens: - USDC - L1 - USDC.e - L2 - LxLy USDC `![](../../img/cdk/stb-1.png)` The diagram above illustrates the following flow: 1. A user initiates a USDC deposit from L1 to L2. 2. Instead of being deposited directly to the unified bridge, the USDC is deposited into the STB L1Escrow contract. 3. The STB L1Escrow locks the USDC and passes a message to the unified messenger containing the userâ€™s address and amount of USDC being bridged. 4. The Messenger contract validates the message and then sends it to the STB L2Escrow. 5. The STB L2Escrow receives the message and mints USDC.e from the L2Token contract. 6. The USDC.e is sent to the user's address on L2. Native token conversion With the introduction of STB, there are now two ways to deposit tokens to an L2 CDK chain, and two distinct resultant tokens: - The STB flow involves locking tokens in the L1Escrow contract, which is followed by the minting of USDC.e on L2. - On the other hand, if the tokens are deposited directly into the LxLy bridge contract, it results in the minting of LxLy USDC on L2. The L2TokenConverter facilitates conversion between USDC.e and LxLy USDC. The way it works is by locking (Deposit function) either of the tokens in the converter contract, and then sending (Withdraw function) the equivalent amount of the other token to the user's wallet. Using the STB contracts The STB contracts grant chain operators control over the token backing in the escrow contracts for all the chains that connect to the Polygon ecosystem via the AggLayer. !!! warning Polygon Labs does not manage or facilitate the staking of the assets locked in the escrow contracts. Any decision to implement any staking strategies and contribute to network security, or otherwise, is completely at the discretion of the chain operators. Bridging tokens to L2 CDK chains using the STB contracts is ideal for the following use cases: - Tokens that need to implement staking/restaking or other mechanisms, or strategies. - Tokens that need to implement custom L2 functionality. - Tokens that possess native issuance capabilities. Want to start testing with STB? The contracts are still being audited, but are ready to use on testnets, and can be found here: https://github.com/pyk/zkevm-stb. # intro-t1-prover.md: The Polygon type 1 prover is a ZK-EVM proving component capable of generating proofs for Ethereum blocks. It has been developed in collaboration with the Toposware team. !!! info The Polygon type 1 prover is not yet ready for full implementation into a CDK stack. Get started If you want to get up and running quickly, follow the how to deploy the type 1 prover guide. !!! warning Throughout this section, we refer to ZK-EVM chains in a general sense and this should not be confused with Polygon's zkEVM product which is a specific example of a ZK-EVM. Type definitions The emergence of various ZK-EVMs ignited the debate of how 'equivalent' is a given ZK-EVM to the Ethereum virtual machine (EVM). Vitalik Buterin has since introduced some calibration to EVM-equivalence in his article, "The different types of ZK-EVMs". He made a distinction among five types of ZK-EVMs, which boils down to the inevitable trade-off between Ethereum equivalence and the efficacy of the zero-knowledge proving scheme involved. For brevity, we refer to this proving scheme as the zk-prover or simply, prover. The types, as outlined by Vitalik, are as follows; - Type 1 ZK-EVMs strive for full Ethereum-equivalence. These types do not change anything in the Ethereum stack except adding a zk-prover. They can therefore verify Ethereum and environments that are exactly like Ethereum. - Type-2 ZK-EVMs aim at full EVM-equivalence instead of Ethereum-equivalence. These ZK-EVMs make some minor changes to the Ethereum stack with the exception of the Application layer. As a result, they are fully compatible with almost all Ethereum apps, and thus offer the same UX as with Ethereum. - Type-2.5 ZK-EVMs endeavor for EVM-equivalence but make changes to gas costs. These ZK-EVMs achieve fast generation of proofs but introduces a few incompatibles. - Type-3 ZK-EVMs seek to be EVM-equivalent but make a few minor changes to the Application layer. These type of ZK-EVMs achieve faster generation of proofs, and are not compatible with most Ethereum apps. - Type-4 ZK-EVMs are high-level-language equivalent ZK-EVMs. These type of ZK-EVMs take smart contract code written in Solidity, Vyper or other high-level languages and compile it to a specialized virtual machine and prove it. Type-4 ZK-EVMs attain the fastest proof generation time. The figure below gives a visual summary of the types, contrasting compatibility with performance. !Figure: ZK-EVM types Ultimately, choosing which type of ZK-EVM to develop involves a trade-off between EVM-equivalence and performance. The challenge this poses for developers who favor exact Ethereum-equivalence is to devise ingenious designs and clever techniques to implement faster zk-provers. Vitalik mentions one mitigation strategy to improve proof generation times: cleverly engineered, and massively parallelized provers. # t1-architecture.md: The Polygon type 1 prover is designed for efficient implementation of STARK proofs and verification of Ethereum transactions. It achieves efficiency by restricting the Algebraic Intermediate Representation (AIR) to constraints of degree 3. The execution trace needed to generate a STARK proof can be assimilated to a large matrix, where columns are registers and each row represents a view of the registers at a given time. From the initial register values on the first row to the final one, validity of each internal state transition is enforced through a set of dedicated constraints. Generating the execution trace for a given transaction unfortunately yields a considerable overhead for the prover. A naÃ¯ve design strategy would be to utilize a single table, which is solely dedicated to the entire EVM execution. Such a table would have thousands of columns, and although it would be a highly sparse matrix, the prover would treat it as fully dense. Modular design strategy Since most of the operations involved in the EVM can be independently executed, the execution trace is split into separate STARK modules, where each is responsible for ensuring integrity of its own computations. These STARK modules are: - Arithmetic module handles binary operations including ordinary addition, multiplication, subtraction and division, comparison operations such as 'less than' and 'greater than', as well as ternary operations like modular operations. - Keccak module is responsible for computing a Keccak permutation. - KeccakSponge module is dedicated to the sponge construction's 'absorbing' and 'squeezing' functions. - Logic module specializes in performing bitwise logic operations such as AND, OR, or XOR. - Memory module is responsible for memory operations like reads and writes. - BytePacking module is used for reading and writing non-empty byte sequences of length at most 32 to memory. Although these smaller STARK modules are different and each has its own set of constraints, they mostly operate on common input values. In addition to the constraints of each module, this design requires an additional set of constraints in order to enforce that these common input values are not tampered with when shared amongst the various STARK modules. For this reason, this design utilizes Cross-table lookups (CTLs), based on a logUp argument designed by Ulrich HabÃ¶ck, to cheaply add copy-constraints in the overall system. The Polygon type 1 prover uses a central component dubbed the CPU to orchestrate the entire flow of data that occurs among the STARK modules during execution of EVM transactions. The CPU dispatches instructions and inputs to specific STARK modules, as well as fetches their corresponding outputs. Note here that â€œdispatchingâ€ and â€œfetchingâ€ means that initial values and final values resulting from a given operation are being copied with the CTLs to and from the targeted STARK module. Prover primitives We now look at the cryptographic primitives used to engineer the Polygon type 1 prover, which is a custom-built prover capable of tracing, proving, and verifying the execution of the EVM through all state changes. The proving and verification process is made possible by the zero-knowledge (ZK) technology. In particular, a combination of STARK[^1] and SNARK[^2], proving and verification schemes, respectively. STARK for proving The Polygon type 1 prover implements a STARK proving scheme, a robust cryptographic technique with fast proving time. Such a scheme has a proving component, called the STARK prover, and a verifying component called the STARK verifier. A proof produced by the STARK prover is referred to as a STARK proof. The process begins with constructing a detailed record of all the operations performed when transactions are executed. The record, called the execution trace, is then passed to a STARK prover, which in turn generates a STARK proof attesting to correct computation of transactions. Although STARK proofs are relatively big in size, they are put through a series of recursive SNARK proving, where each SNARK proof is more compact than the previous one. This way the final transaction proof becomes significantly more succinct than the initial one, and hence the verification process is highly accelerated. Ultimately, this SNARK proof can stand alone or be combined with preceding blocks of proofs, resulting in a single validity proof that validates the entire blockchain back from genesis. Plonky2 SNARK for verification The Polygon type 1 prover implements a SNARK called Plonky2, which is a SNARK designed for fast recursive proofs composition. Although the math is based on TurboPLONK, it replaces the polynomial commitment scheme of PLONK with a scheme based on FRI. This allows encoding the witness in 64-bit words, represented as field elements of a low-characteristic field. The field used, denoted by $\mathbb{F}_p$, is called Goldilocks. It is a prime field where the prime p is of the form $p = 2^{64} - 2^{32} + 1$. Since SNARKs are succinct, a Plonky2 proof is published as the validity proof that attests to the integrity of a number of aggregated STARK proofs. This results in reduced verification costs. This innovative approach holds the promise of a succinct, verifiable chain state, marking a significant milestone in the quest for blockchain verifiability, scalability, and integrity. It is the very innovation that plays a central role in the Polygon type 1 prover. !!! info "Further reading" - The STARK modules, which are also referred to as STARK tables, have been documented in the Github repo here. - We have documented the CPU component while the CPU logic documentation can be found in the repo. - In order to complete the STARK framework, read more about the cross-table lookups (CTLs) and the CTL protocol and range-checks. - Details on Merkle Patricia tries and how they are used in the Polygon type 1 prover can be found here. Included are outlines on the prover's internal memory, data encoding and hashing, and prover input format. [^1]: STARK is short for Scalable Transparent Argument of Knowledge [^2]: SNARK is short for Succinct Non-interactive Argument of Knowledge. # t1-cpu-component.md: The CPU is the central component of the Polygon type 1 prover. Like any central processing unit, it reads instructions, executes them, and modifies the state (registers and the memory) accordingly. Other complex instructions, such as Keccak hashing, are delegated to specialized STARK tables. This section briefly presents the CPU and its columns. However, details on the CPU logic can be found here. CPU flow CPU execution can be decomposed into two distinct phases; CPU cycles, and padding. This first phase of the CPU execution is a lot bulkier than the second, more so that padding comes only at the end of the execution. CPU cycles In each row, the CPU reads code at a given program counter (PC) address, executes it, and writes outputs to memory. The code could be kernel code or any context-based code. Executing an instruction therefore results in modifying registers, possibly performing memory operations, and updating the program counter (PC). In the CPU cycles phase, the CPU can switch between contexts corresponding to different environments depending on calls made. Context 0 refers to the kernel, which handles initialization and termination before and after executing a transaction. - Initialization involves input processing, transaction parsing, and transaction trie updating. - While termination includes receipt creation and final trie checks. Subsequent contexts are created when executing user code. Syscalls, which are specific instructions written in the kernel, may be executed in a non-zero user context. They don't change the context but the code context, which is where the instructions are read from. Padding At the end of any execution, the length of the CPU trace is padded to the next power of two. When the program counter reaches the special halting label in the kernel, execution halts. And that's when padding should follow. There are special constraints responsible for ensuring that every row subsequent to execution halting is a padded row, and that execution does not automatically resume. That is, execution cannot resume without further instructions. CPU columns We now have a look at CPU columns as they relate to all relevant operations being executed, as well as how some of the constraints are checked. These are the register columns, operation flags, memory columns, and general columns. Registers - $\texttt{context}$: Indicates the current context at any given time. So, $\texttt{context}\ 0$ is for the kernel, while any context specified with a positive integer indicates a user context. A user context is incremented by 1 at every call. - $\texttt{codecontext}$: Indicates the context in which the executed code resides. - $\texttt{programcounter}$: The address of the instruction to be read and executed. - $\texttt{stacklen}$: The current length of the stack. - $\texttt{iskernelmode}$: A boolean indicating whether the kernel is on or not. The kernel is a privileged mode because it means kernel code is being executed, and thus privileged instructions can be accessed. - $\texttt{gas}$: The amount of gas used in the prevailing context. It is eventually checked if it is below the current gas limit. And must fit in 32 bits. - $\texttt{clock}$: Monotonic counter which starts at 0 and is incremented by 1 at each row. It is used to enforce correct ordering of memory accesses. - $\texttt{opcodebits}$ These are 8 boolean columns, indicating the bit decomposition of the opcode being read at the current PC. Operation flags Operation flags are boolean flags indicating whether an operation is executed or not. During the CPU cycles phase, each row executes a single instruction, which sets one and only one operation flag. Note that no flag is set during padding. The decoding constraints ensure that the flag set corresponds to the opcode being read. There is no 1-to-1 correspondence between instructions and flags. For efficiency, the same flag can be set by different, unrelated instructions. - Take $\texttt{eqiszero}$ as an example. It represents both the $\texttt{EQ}$ and $\texttt{ISZERO}$ instructions. When there is a need to differentiate them in constraints, they get filtered by their respective opcode. This is possible because the first bit of $\texttt{EQ}$'s opcode is 0, while that of $\texttt{ISZERO}$'s opcode is 1. $\texttt{EQ}$ can therefore be filtered with the constraint: $$ \texttt{eqiszero (1 - opcodebits[0])} $$ and $\texttt{ISZERO}$ with: $$ \texttt{eqiszero opcodebits[0]} $$ Memory columns The CPU interacts with the EVM memory via its memory channels. At each row, a memory channel can execute a write, a read, or be disabled. A full memory channel is composed of the following: - $\texttt{used}$: Boolean flag. If it's set to 1, a memory operation is executed in this channel at this row. If it's set to 0, no operation is executed but its columns might be reused for other purposes. - $\texttt{isread}$: Boolean flag indicating if a memory operation is a read or a write. - $3\ \texttt{address}$ columns. A memory address is made of three parts: $\texttt{context}$, $\texttt{segment}$ and $\texttt{virtual}$. - $8\ \texttt{value}$ columns. EVM words are 256 bits long, and they are broken down in 8 32-bit limbs. The last memory channel is a partial channel. It doesn't have its own $\texttt{value}$ columns but shares them with the first full memory channel. This allows saving eight columns. General columns There are eight (8) shared general columns. Depending on the instruction, they are used differently: - $\texttt{Exceptions}$: When raising an exception, the first three general columns are the bit decomposition of the exception code. These codes are used to jump to the correct exception handler. - $\texttt{Logic}$: For $\texttt{EQ}$ and $\texttt{ISZERO}$ operations, it is easy to check that the result is 1 if $\texttt{input0}$ and $\texttt{input1}$ are equal. It is more difficult to prove that, if the result is 0, the inputs are actually unequal. In order to prove this, each general column must contain the modular inverse of $(\texttt{input0}i - \texttt{input1}i)$ for each limb i, or 0 if the limbs are equal. Then, the quantity $\texttt{general}i (\texttt{input0}i - \texttt{input1}i)$ will be 1 if and only if $\texttt{general}i$ is indeed the modular inverse, which is only possible if the difference is non-zero. - $\texttt{Jumps}$: For jumps, we use the first two columns: $\texttt{shouldjump}$ and $\texttt{condsumpinv}$. The $\texttt{shouldjump}$ column determines whether the EVM should jump: it's 1 for a JUMP, and $\texttt{condition} \neq 0$ for a JUMPI. To check if the condition is actually non-zero for a JUMPI, $\texttt{condsumpinv}$ stores the modular inverse of $\texttt{condition}$, or 0 if it's zero. - $\texttt{Shift}$: For shifts, the logic differs depending on whether the displacement is lower than 2^{32} or not. That is, if it fits in a single value limb. To check if this is not the case, we check if at least one of the seven high limbs is not zero. The general column $\texttt{highlimbsuminv}$ holds the modular inverse of the sum of the seven high limbs, and is used to check it's non-zero like the previous cases. Contrary to the logic operations, we do not need to check limbs individually: each limb has been range-checked to 32 bits, meaning that it's not possible for the sum to overflow and be zero if some of the limbs are non-zero. - $\texttt{Stack}$:

$\texttt{stackinv}$, $\texttt{stackinvaux}$, $\texttt{stackinvaux2}$ are used by popping-only and pushing-only instructions. The popping-only instruction uses the Stack columns to check if the Stack is empty after the instruction. While the pushing-only instruction uses the Stack columns to check if the Stack is empty before the instruction. $\texttt{stacklenboundsaux}$ is used to check that the Stack doesn't overflow in user mode. The last four columns are used to prevent conflicts with other general columns. See the Stack Handling subsection of this document for more details.

t1-ctl-protocol.md: Since each STARK module is like a small state machine where its state transition can be uniquely captured in the form of a table, called the execution trace, the STARK modules are henceforth also referred to as STARK tables. See the repo for further details of each table here. For each STARK table, ordinary STARK proof and verification are used to check if all state transitions occurring in the module satisfies stipulated constraints. However, there are input and output values shared among the tables. These values need to be checked for possible alterations while being shared among tables. For this purpose, cross-table lookups (CTLs) are used in verifying that the shared values are not tampered with. How CTLs work The CTL protocol is based on the logUP argment, which works similar to how range-checks work. Range-checks are discussed in a subsequent document to this one. Example (CTL) Consider the following scenario as an example. Suppose STARK $S2$ requires an operation -- say Op -- that is carried out by another STARK $S1$. Then $S1$ writes input and output values of Op in its own table, and provides these two values as inputs to STARK $S2$. Now, STARK $S2$ also writes the input and output values in its rows, and the table's constraints check if Op is carried out correctly. However, one still needs to ensure that the input and output values in $S1$ are the same as those shared with $S2$. In other words, and in cases where the $S1$ and $S2$ tables have many rows individually, one needs to ensure that all the rows of the $S1$ table involving Op appear amongst the rows of the $S2$ table. Note that, for the sake of efficiency, the $S1$ and $S2$ tables are first reduced to the input and output columns. And thus, this check is tantamount to ensuring that the rows of the $S1$ table involving Op are but permutations of the rows of $S2$ that carry out Op. !Figure: CTL permutation check How the CTL proof works As outlined in the above example, verifying that shared values among STARK tables are not tampered with amounts to proving that rows of reduced STARK tables are permutations of each other. The proof therefore is achieved in three steps; - Filtering rows of interest in both STARK tables. - Computing column-wise 'running sums'. - Checking correct construction and equality of 'running sums'. Table filtering Define filters f^1 and f^2 for STARK tables $S1$ and $S2$, respectively, such that $$ \begin{equation} f^i (rowj) = \begin{cases} 1, & \text{if $rowj$ carries out Op}.\\ 0, & \text{otherwise}. \end{cases} \end{equation} $$ for $i \in \{ 0, 1 \}$ and $0 < j < ni$, where ni is Si's number of rows. So, each filter f^i evaluates to either 0 or 1. Next create subtables $S1'$ and $S2'$ of STARK tables $S1$ and $S2$, respectively, such that $f^i(rowj) = 1$ for all $rowj$ of Si'. !!!note Columns $\{ c^{i,j} \}$ of the filtered subtables Si' only contains values that must be identical (these are shared input and output values.) Let α and β be random challenges which in an interactive setting the verifier sends to the prover. Filters are limited to (at most) degree 2 combinations of columns. Computing running sums For each $i \in \{0,1\}$, let $\{ c^{i,j} \}$ denote the columns of Si'. Define a running sum Zi^{Si} for Si' as follows, $$ Z{n-1}^{Si} = \frac{1}{\sum{j=0}^{m-1}\alpha^j \cdot c{n-1}^{i,j} + \beta} $$ and $$ Zl^{Si} = Z{l+1}^{Si} - fl^i \cdot \frac{1}{\sum{j=0}^{m-1}\alpha^j \cdot cl^{i,j} + \beta} $$ for $0 < l < n-1$. Note that Zl^{Si} is computed backwards. i.e., It starts with $Z{n-1}^{Si}$ and goes down to $Z0^{Si}$ as the final sum. Checking running sums After computing running sums, check equality of the final sums $Z0^{S1} =?\ Z0^{S2}$ and whether the running sums were correctly constructed. The above three steps turn the CTL argument into a LogUp lookup argument, where - the STARK table $S1'$ is the looking table - the STARK table $S2'$ is the looked table which checks for equality between $S1'$ and $S2'$. CTL protocol summary The cross-table protocol can be summarized as follows. For any STARK table S, the prover: - Constructs the looking sums, which are the running sums $\{Zj^l\}$ for each table looking into S. - Constructs the looked sum, which is the running sum Z^S for S. - Sends all the final values $\{Z{j,0}^l\}$ and $Z0^S$ to the verifier. - Sends a commitment to the looking sums $\{Z{j}^l\}$ and the looked sum Z^S to the verifier. On the other side, and for the same STARK table S, the verifier: - Computes the sum $Z = \sumj Z{j}^l$. - Checks equality, $Z =?\ Z0^S$. - Checks whether each of the running sums $Z{j}^l$ and Z^S were correctly constructed.

t1-design-challenge.md: The EVM wasn't designed with zero-knowledge proving and verification in mind, and this makes the design of an efficient type 1 prover extremely challenging. Some of the challenges stem from the way the EVM is implemented. Here are some of the discrepancies that occur when deploying the most common zero-knowledge primitives to the EVM. Word size The native EVM word size is 256 bits long, whereas the chosen SNARK Plonky2, operates internally over 64-bit field elements. Matching these word sizes requires a work-around where word operations are performed in multiples of smaller limbs for proper handling internally. This unfortunately incurs overheads, even for simple operations like the ADD opcode. Supported fields Selecting a field for the most efficient proving scheme can become complicated. Ethereum transactions are signed over the secp256k1 curve, which involves a specific prime field $\mathbb{F}p$, where $p = 2^{256} - 2^{32} - 2^9 - 2^8 -2^7 - 2^6 - 2^4 - 1$. The EVM also supports precompiles for BN254 curve operations, where the computations are carried out in an entirely different field arithmetic. This adds a major overhead when it comes to proving modular arithmetic, as there is a need to deal with modular reductions in the field of the proving system. Such incongruous modular arithmetic is not uncommon. Recursive proving schemes like Halo resorted to utilising two pairing-friendly elliptic curves where proving and verification are instantiated in two different field arithmetics. Other curves, such as the pairing-friendly BLS12-381 popularly used in recursive proving systems, are yet to be EVM-supported in the form of precompiled contracts. Hash functions The EVM uses Keccak as its native hash function both for state representation and arbitrary hashing requests, through the Keccak256 opcode. While Keccak is fairly efficient on a CPU, since Plonky2 implements polynomials of degree 3, Keccak operations would need to be expressed as constraints of degree 3. This results in an extremely heavy Algebraic Intermediate Representation (AIR) compared to some of the most recent STARK-friendly hash functions, tailored specifically for zero-knowledge proving systems. Although the EVM supports precompiles of hash functions such as SHA2-256, RIPEMD-160, and Blake2f, they are all quite heavy for a ZK proving system. State representation Ethereum uses Merkle Patricia Tries with RLP encoding. Both of these are not zero-knowledge-friendly primitives, and incur huge overheads on transaction processing within a ZK-EVM context.

t1-rangechecks.md: Tables often deal with 256-bit words which are split into 16-bit limbs. This helps to avoid field overflows. Range-checks are used for examining integrity of values in these 16-bit limbs. What to range-check? The idea here is to range-check every field element pushed into the Stack, as well as every memory writes. That is, Range-checking the PUSH and MSTORE opcodes. Other range-checks are: - Pushes and memory writes for MSTORE32BYTES, range-checked in the "BytePackingStark". - Syscalls, exceptions and prover inputs are range-checked in "ArithmeticStark". - Inputs and outputs of binary and ternary arithmetic operations are range-checked in "ArithmeticStark". - Inputs' bits of logic operations are checked to be either 1 or 0 in "LogicStark". Since â€œLogicStarkâ€ only deals with bitwise operations, it is sufficient to range-check outputs. - Inputs to Keccak operations are range-checked in â€œKeccakStarkâ€. The output digest is written as bytes in â€œKeccakStarkâ€. Those bytes are used to reconstruct the associated 32-bit limbs checked against the limbs in â€œCpuStarkâ€. This implicitly ensures that the output is range-checked. What not to range-check Some operations do not require range-checks, including the following: - MSTOREGENERAL, which writes values read from Stack. Therefore, the written values were already range-checked by previous pushes. - EQ, which reads two -- already range-checked -- elements on the Stack, and checks their equality. The output is either 0 or 1, and therefore need not be range-checked. - NOT, which reads one -- already range-checked -- element. The result is constrained to be equal to $\texttt{0xFFFFFFFF} - \texttt{input}$, which implicitly enforces the range-check. - PC, the Program Counter, which cannot be greater than 2^{32} in user mode. Indeed, the user code cannot be longer than 2^{32}, and jumps are constrained to be JUMP destinations, JUMPDESTs. Moreover, when in kernel mode, every JUMP's destination is a location within the kernel, and the kernel code is smaller than 2^{32}. These two points implicitly enforce range-check on PC's. - GETCONTEXT, DUP, and SWAP, all read and push values that were already written in memory. These pushed values were therefore already range-checked. Note that range-checks are performed on the range $[0, 2^{16} - 1]$, so as to limit the trace length. Lookup argument Enforcement of range-checks leverages LogUp, a lookup argument introduced by Ulrich HÃ¤bock. Given a looking table $s = (s1, ..., sn)$ and a looked table $t = (t1, ..., tm)$, the goal is to prove that $$ \text{for all}\ 1 \leq i \leq n,\ \text{there exists}\ 1 \leq j \leq r\ \text{ such that }\ si = tj $$ In our case, $t = (0, .., 2^{16} - 1)$ and s is composed of all the columns in each STARK that must be range-checked. The LogUp paper explains that proving the previous assertion is equivalent to proving that there exists a sequence $\{lj \}$ such that: $$ \sum{i=1}^n \frac{1}{X - si} = \sum{j=1}^r \frac{lj}{X-tj} \tag{1} $$ The values in the looking table $s = (s1, ..., sn)$, can be stored in columns each of length n. And if these columns are c in number, the above equality becomes: $$ \sum{k=1}^c \sum{i=1}^n \frac{1}{X - si^k} = \sum{j=1}^r \frac{lj}{X-tj} \tag{2} $$ The multiplicity mi of each value ti is defined as the number of times ti appears in the looking table $s = (s1, ..., sn)$. In other words, mi is the cardinality of a set, given by: $$ mi = \big|\{ sj \in s\ |\ sj = ti \}\big| $$ Multiplicities of the $\{ tj \}$ form a sequence $\{ mj \}$ and thus proves existence of the required $\{ lj \}$ sequence of Equation 1 above. This means Equation 2 can be rewritten as: $$ \sum{k=1}^c \sum{i=1}^n \frac{1}{X - si^k} = \sum{j=1}^r \frac{mj}{X-tj} $$ For each random challenge α, provided by the verifier, proving the lookup argument amounts to checking this equation: $$ \sum{k=1}^c \sum{i=1}^n \frac{1}{\alpha - si^k} = \sum{j=1}^r \frac{mj}{\alpha-tj} $$ However, this yields a high degree equation. HÃ¤bock suggests circumventing this issue by providing helper columns $\{hi\}$ and d , such that at any given row i: $$ \begin{aligned} &hi^k = \frac{1}{\alpha + si^k }\ \text{ for all }\ 1 \leq k \leq c \\ &di = \frac{1}{\alpha + ti} \end{aligned} $$ The h helper columns can be batched together to save columns. At most $\texttt{constraint\degree} - 1$ helper functions can be batched together. In our case, they are batched 2 by 2. For row i, we therefore obtain: $$ hi^k = \frac{1}{\alpha + si^{2k}} + \frac{1}{\alpha + si^{2k+1}}\ \text{ for all }\ 1 \leq k \leq c/2 $$ If the number of column c is odd, we have one extra helper column: $$ hi^{c/2+1} = \frac{1}{\alpha + si^{c}} $$ We henceforth assume c to be even. Now, let g be a generator of a subgroup of order n. Extrapolate h, m, and d in order to get polynomials such that, $$ f(g^i) = fi \ \text{ for }\ f \in \{h^k, m, g\} $$ Define the following polynomial: $$ Z(x) := \sum{i=1}^n {\huge[}\sum{k=1}^{c/2} h^k(x) - m(x) d(x){\huge]} $$ Constraints Given the above definitions and a challenge α, the following constraints can be used to determine whether the assertion holds true: $$ \begin{aligned} &Z(1) = 0 \\ &Z(g \alpha) = Z(\alpha) + \sum{k=1}^{c/2} h^k(\alpha) - m(\alpha) d(\alpha) \end{aligned} $$ It still remains to ensure that h^k is well constructed for all $1 \leq k \leq c/2$: $$ h(\alpha)^k \cdot (\alpha + s{2k}) \cdot (\alpha + s{2k+1}) = (\alpha + s{2k}) + (\alpha + s{2k+1}) $$ Note that, if c is odd, then there is one unbatched helper column $\{h^{{(c/2)}+1}\}$ for which we need a last constraint: $$ h(\alpha)^{{(c/2)}+1} \cdot (\alpha + s{c}) = 1 $$ Finally, the verifier needs to ensure that the looked table $t = (t1, ..., tm)$, was correctly computed. In each STARK, t is computed starting from 0 and adding at most 1 at each row. This construction is constrained as follows: $$ \begin{aligned} \text{1. }\quad &t(1) = 0 \\ \text{2. }\quad &(t(g^{i+1}) - t(g^{i})) \cdot ((t(g^{i+1}) - t(g^{i})) - 1) = 0 \\ \text{3. }\quad &t(g^{n-1}) = 2^{16} - 1 \end{aligned} $$

testing-and-proving-costs.md: Testing the prover Find a parser and test runner for testing compatible and common Ethereum full node tests against the Polygon type 1 prover here. The prover passes all relevant and official Ethereum tests. Proving costs Instead of presenting gas costs, we focus on the cost of proving EVM transactions with the Polygon type 1 prover. Since the prover is more like a 'CPU' for the EVM, it makes sense to look at proving costs per VM instance used, as opposed to TPS or other benchmarks. Consider the table below for prices of GCP's specific instances, taken from here, and webpage accessed on the 29th January, 2024. !Figure: GCP's vm instance price Take the example of a t2d-standard-60 GCP instance, where each vCPU has 4GB memory, based on GCP's Spot prices: - 0.00346 USD / vCPU hour - 0.000463 USD / GB hour We obtain the following hourly cost, $(60 \times 0.00346) + (240 \times 0.000463) = 0.31872$ USD per hour. The total cost per block is given by: $\texttt{(Proving time per hr)} \times 0.31872$ USD. The table below displays proving cost per transaction per hour.

Block number	Transactions	Gas	Proof time (minutes)	Total cost (USD)	Cost per tx (USD)
17106222	105	10,781,405	44.17	$0.235	$0.0022
17095624	163	12,684,901	78.12	$0.415	$0.0025
17735424	182	16,580,448	100.52	$0.534	$0.0029

admin-upgradeability.md: Admin upgradeability As L2s work through the stages of training wheels to become fully decentralized, chains that opt in to the AggLayer implement shared security mechanisms with other AggLayer chains including the Polygon zkEVM to ensure the safety of users. Chains opted into the AggLayer share the following upgradeability controls: 1. The security council (contract address) that can be used to trigger the emergency state which can pause bridge functionality, prevent smart contract upgrades, or stop the sequencer from sequencing batches. 2. The admin role (contract address) that can perform upgrades to patch bug fixes or add new features to the system by upgrading smart contracts with a 10-day waiting period (unless emergency state is active). Further reading - zkEVM protocol upgradability. - zkEVM admin role and governance. - zkEVM upgrade process. - zkEVM security council. - zkEVM emergency state. - L2Beat - Polygon zkEVM.

architecture.md: Architecture While each chain built with the CDK is unique, they all share a common high-level architecture. Before diving into the specifics of how transactions are processed, it is helpful to first understand the role of each component in the system. Below is the high-level architecture of a chain built with the CDK, showing how transactions sent by users are processed and finalized on the L1: !CDK Architecture Users Chains built with the CDK are EVM-compatible by default. Although the type of ZK-EVM you choose to implement is customizable, CDK chains are designed to be compatible with existing Ethereum tools and libraries. This means both users and developers can use the same wallets (such as MetaMask) and libraries (such as Ethers.js) to interact with CDK-built chains as they do with Ethereum. The process for submitting transactions is the same as on Ethereum, using the same JSON-RPC interface. Transactions are submitted directly to the L2 and go into a pending transaction pool. Sequencer The sequencer is responsible for two vital tasks in the system: 1. Executing transactions submitted by L2 users. 2. Sending batches of transactions to the L1 smart contract. The sequencer reads transactions from the pending transaction pool and executes them on the L2, effectively updating the state of the L2 and providing this information back to the user. Once this process is complete (typically in a matter of seconds), users are free to continue interacting with the L2 as if the transaction was finalized. In the background, the sequencer periodically creates batches of transactions and sends multiple batches of transactions to the L1 smart contract in a single transaction. L1 smart contracts Multiple smart contracts, deployed on the L1 (Ethereum), work together to finalize transactions received from the L2 on the L1. Typically there is a main rollup smart contract that is responsible for: 1. Receiving and storing batches of transactions from the L2 (depending on the design of the L2, it may not use Ethereum for data availability). 2. Receiving and verifying ZK-proofs from the aggregator to prove the validity of the transactions. Aggregator and prover The aggregator is responsible for periodically reading batches of L2 transactions that have not been verified yet, and generating ZK-proofs for them to prove their validity. To do this, the aggregator sends the batches of transactions to a prover. The prover generates ZK proofs and sends them back to the aggregator, which then posts the proof back to the L1 smart contract. Further reading - zkEVM architecture overview

blocks.md: Batches, blocks, and transactions The following definitions are key to understanding how transactions are handled on L2s built with the CDK: - Transaction: A signed instruction to perform an action on the blockchain. - Block: A group of transactions and a hash of the previous block in the chain. - Batch: A group of many transactions from multiple blocks. Transactions are included in blocks, and these blocks fill batches, which are then sequenced. See the figure below to best understand how these elements relate to each other. !Batches, blocks, transactions Transaction A transaction is a cryptographically signed instruction from an account to update the state of the blockchain. Let's take a look at a real transaction in the Polygon zkEVM (which is in a way a CDK rollup), and inspect how the transaction is recorded in the explorer as part of a block, then a batch, and ultimately in a sequence. Consider the Polygon zkEVM transaction with the transaction hash, 0xdd ... 6ef8, which performs a Simple Swap function call, and is included in block number 12952601 on the L2. !Transaction with block number Block Each block must include the hash of the previous block, along with multiple transactions, to establish a link to the block before it. Continuing with the above transaction, identified by the hash 0xdd ... 6ef8 and contained in block 12952601, we observe that this block contains 2 transactions in total. Also, as depicted in the figure below, block 12952601 is in turn contained in batch 2041736. !Block and batch Batch Batches contain multiple transactions from multiple blocks. The two transactions from our example block 12952601 are included in batch 2041736, which contains 10 transactions in total. This means batch 2041736 includes the two transactions from block 12952601 as well as eight transactions from other blocks. As observed in the figure below, the presence of the Sequence Tx Hash field associated indicates that the batch has been sent to Ethereum along with other batches in a single transaction. !Batch of transactions Further inspection of the transactions in the batch, as depicted in the figure below, reveals that: - Our transaction example, with hash 0xdd ... 6ef8, is included in this batch. - The batch contains several transactions from different blocks. !Transaction found inside batch Since Polygon zkEVM is a rollup (that is, it uses Ethereum for data availability), it sends an array of batches to Ethereum, by providing the array as an argument to the sequenceBatches() function of a smart contract on Ethereum. !Sequence Transaction We can inspect the Sequence Tx Hash transaction to ascertain if batch 2041736, containing our original transaction example, was indeed part of the argument to the sequenceBatches() function. In this case, batch 2041736 happens to be the last batch to be sequenced, as depicted in the figure below. !Last batch sequenced Further reading - Blocks in the zkEVM Etrog upgrade.

bridging.md: Bridging Bridges are a fundamental component of L2s that allow users to deposit and withdraw assets to and from your chain. CDK-built chains come with a built-in bridge service and customizable UI out of the box, with the option to have a standalone LxLy bridge or alternatively opt-in to the AggLayer and use the unified bridge to enable cross-chain L2-to-L2 interoperability. LxLy bridge The LxLy bridge contracts carry out deposit and withdrawal of assets between L2 and L1. Chains looking to run their own bridge infrastructure can choose to deploy a new instance of the LxLy bridge that allows users to move assets (both native and ERC20 tokens) from L1 to the L2 and vice versa. Deploying an individual instance of the LxLy means interoperability with other L2 chains via the AggLayer is not possible. To enable cross-chain interoperability (i.e. L2-to-L2 cross-chain transactions), chains can opt-in to the AggLayer and use the unified bridge. This option is suited to chains that may want to customize how the bridge is managed and operated, or maintain control of the bridge's funds; as the upgradeability of the bridge contracts are managed by the chain operator. !LxLy bridge Unified bridge A single, shared instance of the LxLy bridge, called the unified bridge is available to use for all CDK chains that opt-in to the AggLayer. It is a shared smart contract deployed on Ethereum, responsible for enabling interoperability between chains in the form of cross-chain transactions and L2-to-L2 transfers. Chains that integrate with the unified bridge can benefit from the network effect of the AggLayer, as their chain can therefore access the users and liquidity of other chains that are also part of the AggLayer. This option is suited to chains that want a standard bridging experience and do not require customization of the bridge's operation. The shared bridge is also not directly managed by the chain operator, instead, it shares the governance outlined in the admin upgradeability section. !unified bridge Further reading - Aggregated blockchains: A new thesis. - LxLy bridge. - Unified bridge overview.

gas-fees.md: Gas fees CDK chains have full control over how gas fees are set for users, including what token to use for the native gas fees of the L2 chain, which is set to ETH by default. Gas fees can also be omitted entirely, allowing users to interact with the chain without needing to pay gas fees for transactions and have the fees covered by the chain operator. By default, gas fees on the L2 are determined by a combination of several factors, including the current gas price on Ethereum, the complexity of the submitted transaction, and the current demand on the L2 network itself. Further reading - zkEVM gas fees documentation.

layer2s.md: What is a layer 2 blockchain? Layer 2 (L2) blockchains are scaling solutions, typically built on top of Ethereum (L1) that are designed to increase transaction throughput without sacrificing decentralization or security. While L2s are their own chains, they are considered "extensions" of Ethereum. Users can submit transactions directly to L2 chains, which handle them more efficiently (in terms of cost and speed) than Ethereum. Under the hood, L2s create "batches" of transactions, and periodically submit many batches to Ethereum as a single transaction; potentially including information on thousands of transactions that occurred on the L2 in a single transaction on Ethereum. Typically, L2s deploy smart contracts to Ethereum that handle the verification of these batches, ensuring that the transactions are valid. Since this verification process occurs on Ethereum, it is often said that L2s inherit the security of Ethereum. !L2 batching overview Types of layer 2s L2s come in different shapes and sizes in terms

of their relationship with Ethereum: Each design decision comes with trade-offs in terms of security, scalability, or decentralization. For example, some L2s, such as the Polygon zkEVM send all transaction data to Ethereum, whereas other L2s only send information about the state differences, or choose not to send transaction data to Ethereum; instead relying on different data availability mechanisms. Since storing information on Ethereum is expensive, (see gas and fees), building an L2 chain means making tradeoffs between security, decentralization, and scalability. The CDK provides developers with the tools to make these trade-offs and build a chain that meets their specific needs depending on their use case. Further reading - Ethereum documentation: Layer 2s. - Ethereum documentation: Scaling. # rollup-vs-validium.md: Rollups vs. validiums Layer 2s can differ in how they interact with Ethereum. More specifically, they often differ in what they do with the transaction data (i.e. the information about transactions that occurred on the L2). L2s can be broadly categorized into two types: rollups and validiums. Rollups Rollups use Ethereum as a data availability (DA) layer, meaning they send and store transaction data directly on Ethereum, by providing it within specific parameters of a transaction submitted to a smart contract on the L1. Using Ethereum to store transaction data is generally considered the most secure option for DA as it leverages Ethereum's security and decentralization. However, this approach is costly, as the L2 must pay Ethereum's high gas fees for storing data on the L1, which typically results in higher gas fees for users. Within the rollup category, there are further nuances to storing transaction data on Ethereum. Some rollups post serialized transaction data directly, whereas others post state differences instead. Some rollups use calldata to store transaction data, while others use more recent Ethereum features such as blobs, introduced in EIP-4844. The CDK provides full flexibility to developers to choose what to do with transaction data, including the ability to build rollups that store data on Ethereum as a rollup like the Polygon zkEVM. !!! note Currently, the rollup mode of Polygon CDK does not support blobs (EIP-4844), but this functionality is coming soon. Validiums Validiums do not store transaction data on Ethereum. Instead, they only post ZK-proofs called validity proofs to Ethereum that verify the state of the L2 chain. As an L2, a validium does not pay high gas fees associated with storing data on Ethereum. This approach is more cost-effective than rollups, meaning gas fees are much lower for users. However, validiums are typically considered less secure than ZK-rollups, as they store transaction data off of Ethereum using solutions such as a Data Availability Committee (DAC) or alternative data availability solutions. Alternative DA solutions The CDK supports integration of alternative DA solutions, including solutions such as Avail DA, Celestia, Near DA and more. What's best for you? The method you use to store transaction data should be determined by your specific use case. As a general rule of thumb: - Rollups are more suitable for chains that process high-value transactions where security is the top priority, such as DeFi applications, as they are considered more secure with slightly higher fees and limited throughput. - Validiums are more suitable for chains that process a high volume of transactions where low transaction fees are the top priority, such as gaming or social applications, as they are considered more scalable and offer low fees. !zkEVM Rollup vs Validium Default configuration By default, chains built with the CDK are set up as a validium. For most use cases, the validium option is the more suitable as it offers lower gas fees and higher throughput, while maintaining strong security guarantees provided by the use of validity proofs. Technical comparison Below is a breakdown of the technical differences between a zkEVM rollup and validium: | | Rollup | Validium | | -------------------- | -- | --- | Node type | zkEVM node | Validium node: zkEVM node with validium extensions | | Data availability | On-chain via L1 | Off-chain via a local option, or a DAC + DA node | | Components | zkEVM components\ | zkEVM components\ + PostgreSQL database + on-chain committees | | Contracts | zkEVM smart contracts

- PolygonZkEVM (main rollup contract)
- PolygonZkEVMBridge
- PolygonZkEVMGlobalExitRoot

| Validium-specific DAC contract

- CDKDataCommittee.sol
- CDKValidium.sol

| | Infrastructure | Standard infrastructure | Dedicated infrastructure for data availability layer and DACs | | Tx flow | All transaction data is published on L1 | Validium only publishes the hash of the transaction data to L1. The sequencer sends both the hash and the transaction data to the DAC for verification. Once approved, the hash+signatures are sent to the Consensus L1 contract of the validium protocol. | | Security | High security due to on-chain data availability and zero-knowledge proofs. | Off-chain data availability can affect security if the sequencer goes offline or if DAC members collude to withhold state data. | | Gas fees | High, because all transaction data is stored on Ethereum. | Low, because only the hash of the transaction data is stored on Ethereum. | | Proof generation | Uses Prover to generate proofs of batched transactions for validation. | Uses Prover to generate proofs of batched transactions for validation. | | Final settlement | Transaction batches and their corresponding proofs are added to the Ethereum state. | The hash of transaction data and its proof are added to the Ethereum state, referred to as the consolidated state. | \JSON RPC, Tx pool manager, Sequencer, Etherman, Synchronizer, State DB, Aggregator, Prover Further reading - Ethereum documentation: rollups - Ethereum documentation: validiums # transaction-finality.md: Transaction finality Throughout the transaction lifecycle, transactions progress through three states of finality: 1. Trusted: The transaction has been submitted and executed on the L2. The user receives the result of the transaction and can continue to interact with the L2 chain. 2. Virtual: The transaction has been batched and sequenced, meaning the batch containing the transaction has been sent to Ethereum. However, the ZK-proof to verify the validity of the transaction has not yet been posted and verified on Ethereum. 3. Consolidated: The transaction has been aggregated, meaning a ZK-proof has been generated, posted, and verified on Ethereum to prove the validity of the transaction. The transaction is now considered final at the L1 level, enabling the user to withdraw their funds from the L2 chain back to Ethereum. !Transaction finality Further reading - zkEVM state management. - zkEVM transaction lifecycle. # transaction-lifecycle.md: Transaction lifecycle Transactions on CDK-built chains go through a series of steps to eventually reach finality on Ethereum. Specifically, they go through the following steps: 1. Submitted: The transaction is submitted to the L2. 2. Executed: The transaction is executed on the L2 by the sequencer. 3. Batched: The transaction is included in a batch of transactions. 4. Sequenced: The batch containing the transaction is sent to Ethereum. 5. Aggregated: A ZK-proof is generated, posted, and verified on Ethereum to prove the transaction is valid. Submitted Similar to Ethereum, users submit transactions to a pool of pending transactions on the L2. The transaction is submitted using the same interface as on Ethereum, via JSON-RPC which is implemented by wallets such as MetaMask and developer libraries such as Ethers.js. !User submitting transactions to L2 Executed The sequencer reads transactions from the pending transaction pool and executes them on the L2. Once executed, transactions are added to blocks, then the blocks fill batches, and the sequencerâ€™s local L2 state is updated. Once the state is updated, it is also broadcast to all other zkEVM nodes which provide the transaction information back to the user or application that submitted the transaction. At this point, the transaction appears complete to users, as they are provided with the result of whether the transaction was executed or reverted. Users can continue to interact with the chain with the updated state. !Transaction executed on L2 Batched As a background process, the sequencer is constantly creating batches of transactions. These batches contain multiple transactions from multiple blocks on the L2. One field in the batch data structure is transactions, which contains a bytes representation of the transactions in the batch. This is generated by serializing each transaction in the batch using RLP serialization and then concatenating them together. !Batch of transactions Sequenced Depending on the data availability design choices of the L2, if the L2 is a rollup, the sequencer sends arrays of batches to the L1 smart contract, where they are stored inside the state of the smart contract. !Sequence Batches Aggregated The final step of the transaction lifecycle is to generate a ZK-proof that proves the batch of transactions is valid. Batches of transactions are read by the aggregator which utilizes a prover to generate a ZK-proof that is posted back to Ethereum. !Aggregator posting ZK-proof Further reading - zkEVM transaction lifecycle documentation # zk-vs-optimistic.md: ZK vs. optimistic rollups Rollups can differ in how they ensure the validity of transactions that occur on the L2. They can be broadly categorized into two types: zero-knowledge (ZK) rollups and optimistic rollups. Importantly, the Polygon CDK allows developers to build ZK rollups as they use cryptographic mechanisms (ZK-proofs) for security, which offers several advantages and are more secure than optimistic rollups that rely on the honesty of incentivized actors. Optimistic rollups !!! note The CDK does not support the development of optimistic rollups. Optimistic rollups act on an "innocent until proven guilty" basis, meaning they optimistically assume transactions that occurred on the L2 are valid and do not actively post any validity proofs to Ethereum to prove the validity of transactions. Instead, they rely on a challenge period, a set period of time (typically 7 days) where users (or sometimes a set list of actors) can compute and submit a fault proof (often called a fraud proof) to challenge the results of a rollup transaction. If the fault proof is accepted, meaning there were fraudulent transactions posted to Ethereum, the state of the rollup is updated to reflect the correct state, and the malicious actor (the sequencer) is penalized. The key advantage to optimistic rollups is that they are cost-effective. Because fraud proofs are only rarely submitted in the event of a dispute, the gas fees associated with optimistic rollups are lower than ZK rollups which are regularly posting proofs to Ethereum to prove the validity of transactions. However, there are several disadvantages to this optimistic approach; users cannot withdraw their funds until the challenge period has passed, and the security of the rollup is dependent on the honesty of the actors. That is, at least one honest actor must be actively monitoring the rollup and submitting fault proofs to maintain the integrity of the chain, or the rollup could be compromised. Zero-knowledge rollups ZK rollups act on a "guilty until proven innocent" basis, meaning transactions are only considered valid once an associated validity proof (ZK-proof) is posted and verified on Ethereum to prove the validity of transactions. By using cryptographic mechanisms for security, ZK rollups are generally considered more secure than optimistic rollups as there is no situation where fraudulent transactions can be finalized; unlike optimistic rollups which rely on a challenge period to correct any fraudulent transactions. As validity proofs are posted regularly to Ethereum to prove the validity of transactions (for example, see the Polygon zkEVM trusted aggregator), the gas fees are typically slightly higher than optimistic rollups, as the L2 must pay Ethereum's high gas fees to store data on the L1. ZK rollups have several advantages over optimistic rollups; users can withdraw their funds without waiting for a challenge period to pass, and the security of the rollup is not dependent on the honesty of actors, as the cryptographic mechanisms ensure the validity of transactions. Further reading You can learn more about the differences between ZK and optimistic rollups on the official Ethereum documentation: - Ethereum documentation: ZK Rollups. - Ethereum documentation: Optimistic Rollups. # cli-tool.md: To simplify the process of running and configuring CDK components, Polygon provides a Rust-based CLI tool which is an interface that chain administrators can use to interact with the components. This CLI tool is an entry point for chain administrators to access the CDK system. Installation As the chain admin, you simply need to download the precompiled CDK package binaries. Running the CLI tool !!! info Requirements: Get the binaries, packages and docker images published with each release, here. Commands CDK Here, you need to provide the CDK node configuration file and the genesis file for your desired chain. Usage: cdk Commands: node - Run the cdk-node with the provided configuration erigon - Run cdk-erigon node with the provided default configuration versions - Output the corresponding versions of the components help Print this message or the help of the given subcommand(s) Options: -h, --help - Print help cdk node To run cdk-node use the node subcommand with one of the options mentioned below. Usage: cdk node [OPTIONS] Options: -C, --config - The path to the configuration file [env: CDKCONFIGPATH=] -c, --components - Components to run [env: CDKCOMPONENTS=] -h, --help - Print help Example to run in FEP mode: cdk node --config /etc/cdk/cdk-node-config.toml --components sequence-sender,aggregator Example to run in PP mode: cdk node --config /etc/cdk/cdk-node-config.toml --components rpc,aggsender cdk erigon You can run a cdk-erigon RPC node that syncs to an existing chain using the default parameters. This subcommand is intended for quickly spinning up an RPC node or testing existing chains with default configuration values. In order to fine-tune settings and access all available configuration options, refer to the full cdk-erigon documentation on Erigon configuration. Usage: cdk erigon [OPTIONS] Options: -C, --config - The path to the cdk-node configuration file [env: CDKCONFIGPATH=] -g, --chain - The path to the genesis.json file [env: CDKGENESISPATH=] -h, --help - Print help cdk erigon --config /etc/cdk/cdk-node-config.toml --chain genesis.json cdk versions The above command generates all the required configuration files for cdk-erigon on the fly and runs the node. To print the corresponding versions of the components, run the following command: Usage: cdk versions Options: -h, --help - Print help Example: cdk versions # local-deployment.md: This guide walks you through the process of setting up and deploying a layer 2 CDK blockchain stack on your local machine. The Polygon CDK Kurtosis package allows you to easily customize and instantiate all the components of a CDK chain. It uses the Kurtosis tool to orchestrate the setup of the chain components in Docker containers, with logic defined in Starlark scripts (a Python dialect) which define the step-by-step process of setting up the chain. !!! tip Check out the Polygon Kurtosis docs for more documentation on this stack and how to use it, and if you need to raise an issue or have a question for the team. Prerequisites Hardware/OS - x86-64 architecture. - Minimum 8GB RAM/2-core CPU. - Linux-based OS (or WSL). Software - Docker Engine - version 4.27 or higher for MacOS. - Kurtosis CLI And, optionally, for submitting transactions and interacting with the environment once set up, we are using: - Foundry - yq (v3) - jq - polyon-cli Set up the Kurtosis environment Understanding the deployment steps There are two configuration files which help you understand what happens during a deployment. 1. main.star The main.star file contains the step-by-step instructions for the deployment process. It orchestrates the setup of all the components in sequential order and pulls in any necessary logic from other files. It defines the following steps for the deployment process: | Step number | Deployments | Relevant Starlark code | Enabled by default | |-------------|---|--------------------|| | 1 | Deploy a local layer 1 Ethereum chain | ethereum.star | True | | 2 | Deploy the CDK smart contr on the L1 | deployzkevmcontracts.star | True | | 3 | Deploy the central environment, prover, and CDK erigon or zkEVM node databases | databases.star | 4 | Get the genesis file | n/a | False | | 5 | Deploy the CDK central environment | cdkcentralenvironment.star | True | | 6 | Deploy the CDK erigon package | cdkerigon.star - included in step 4 deployment | True | | 7 | Deploy the bridge infrastructure | cdkbridgeinfrastructure.star | True | | 8 | Deploy the AggLayer | agglayer.star | True | | 9 | Additional services | Explorers, reporting, permissionless zkEVM node | False | | - | Input parser tool to help deployment stages | inputparser.star - deployed immediately | n/a | | - | zkEVM pool manager tool | zkevmpoolmanager.star - deployed with CDK erigon node | n/a | !!! warning - The Kurtosis stack is designed for local testing only. - The prover component is a mock prover and should never be used for production environments. You can customize (or skip) any of the numbered steps by modifying the logic in the respective files. 2. params.yml The params.yml file defines the parameters of the chain and the deployment process. It includes configurations for simple parameters such as the chain ID and more complex configurations such as the gas token smart contract address. You can modify each of these parameters to customize the chain to your specific needs. Run the chain locally 1. In the kurtosis-cdk directory, use the kurtosis run command to deploy the chain on your local machine by executing the main.star script provided with the params.yml configuration file: bash kurtosis run --enclave cdk github.com/0xPolygon/kurtosis-cdk 2. This command typically takes a while to complete and outputs the logs of each step in the deployment process for you to monitor the progress of the chain setup. Once the command is complete, you should see the following output: bash Starlark code successfully run. No output was returned. INFO[2024-10-17T10:56:06+02:00] == INFO[2024-10-17T10:56:06+02:00] || Created enclave: cdk || INFO[2024-10-17T10:56:06+02:00] == Name: cdk UUID: 0fb1ba8e87ad Status: RUNNING == Files Artifacts == ... List of files generated during the deployment process ... === User Services === ... List of services with "RUNNING" status - none should be "FAILED"! ... The default deployment includes cdk-erigon as the sequencer, and cdk-node functioning as the sequence sender and aggregator. You can verify the default versions of these components and the default fork ID by reviewing inputparser.star. You can check the default versions of the deployed components and the default fork ID by looking at inputparser.star. 3. Customize the chain To make customizations to the CDK environment, clone this repo, make any desired configuration changes, and then run: sh Delete all stop and clean all currently running enclaves kurtosis clean --all Run this command from the root of the repository to start the network kurtosis run --enclave cdk . 4. Inspect the chain Get a feel for the entire network layout by running the following command: sh kurtosis enclave inspect cdk Interacting with the chain Now that your chain is running, you can explore and interact with each component. Below are a few examples of how you can interact with the chain. Read/write operations Let's do some read and write operations and test transactions on the L2 with Foundry. 1. To facilitate the operations, export the RPC URL of your L2 to an environment variable called ETHRPCURL with the following command: bash export ETHRPCURL="$(kurtosis port print cdk cdk-erigon-node-001 rpc)" 2. Use cast to view information about the chain, such as the latest block number: bash cast block-number 3. View the balance of an address, such as the pre-funded admin account: bash cast balance --ether 0xE34aaF64b29273B7D567FCFc40544c014EEe9970 4. Send simple transactions to the chain, such as a transfer of some ETH: bash --private-key "0x12d7de8621a77640c9241b2595ba78ce443d05e94090365ab3bb5e19df82c625" cast send --legacy --value 0.01ether 0x0000000000000000000000000000000000000000 Load testing the chain 1. Use the polycli loadtest command to send multiple transactions at once to the chain to test its performance: bash polycli loadtest --rpc-url "$ETHRPCURL" --legacy --private-key "$PK" --verbosity 700 --requests 50000 --rate-limit 50 --concurrency 5 --mode t polycli loadtest --rpc-url "$ETHRPCURL" --legacy --private-key "$PK" --verbosity 700 --requests 500 --rate-limit 10 --mode 2 polycli loadtest --rpc-url "$ETHRPCURL" --legacy --private-key "$PK" --verbosity 700 --requests 500 --rate-limit 3 --mode uniswapv3 Grab some logs Add the service name to the following command to grab the logs you're interested in. bash kurtosis service logs cdk agglayer --follow Open a shell on a service To open a shell to examine a service, add the service name to the following command. bash kurtosis service shell cdk contracts-001 jq . /opt/zkevm/combined.json Viewing transaction finality A common way to check the status of the system is by ensuring that batches are send and verified on the L1 chain. Use cast to view the progression of batches from trusted, virtual, and verified states: bash cast rpc zkevmbatchNumber Latest batch number on the L2 cast rpc zkevmvirtualBatchNumber Latest batch received on the L1 cast rpc zkevmverifiedBatchNumber Latest batch verified or "proven" on the L1 Opening the bridge UI To open the zkevm-bridge interface and bridge tokens across the L1 and L2, run the following command: bash open $(kurtosis port print cdk zkevm-bridge-proxy-001 web-ui) Additional services There are a number of additional services you can add to the stack, including observability applications and other useful tools. See the current list of additional services in the CDK kurtosis additional services documentation. To add an additional service, simply add the name of the service to the params.yml array. For example: yml args: additionalservices: - blockscout - prometheusgrafana To use the additional service, simply add the service to a kurtosis call. For

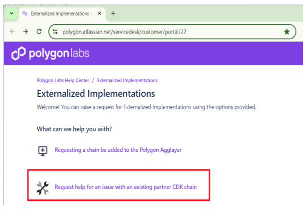
example, to open the Grafana dashboard once set up in params.yml, run the following command: bash open $(kurtosis port print cdk-v1 grafana-001 dashboards) !Panoptichain Dashboard Stopping the chain If you want to stop the chain and remove all the containers, run the following command: bash kurtosis clean --all Going to production While it is possible to run a CDK chain on your own, we strongly recommend getting in touch with the Polygon team directly, or one of our implementation providers for production deployments. Advanced use cases For a list of advanced use cases and documentation explaining how to set them up, please see the list in the Kurtosis CDK stack repo. Further reading - For more information on CDK architecture, components, and how to customize your chain, refer to the CDK architecture documentation. - For detailed how to's, including how to create a native token, check out our how to guides. - For detailed conceptual information on zero-knowledge stacks, check out our concepts documentation. # chain-config.md: To use chains other than the defaults, supply a set of custom configuration files. 1. Ensure the chain name starts with the word dynamic e.g. dynamic-mynetwork. 2. Create the following files for dynamic configs. The examples for Cardona are in zk/examples/dynamic-configs and can be edited as required: - dynamic-{network}-allocs.json - the allocs file. - dynamic-{network}-chainspec.json - the chainspec file. - dynamic-{network}-conf.json - an additional configuration file. - dynamic-{network}.yaml - the run config file for erigon. You can use any of the example yaml files at the root of the repo as a base and edit as required, but ensure the chain field is in the format dynamic-mynetwork and matches the names of the config files above. 3. Put the erigon config file, along with the other files, in the directory of your choice. For example dynamic-mynetwork. !!! tip - If you have allocs in the Polygon format from the original network launch, save this file to the root of the cdk-erigon code base and run go run cmd/hack/allocs/main.go [your-file-name] to convert it to the format needed by erigon. - This creates the dynamic-{network}-allocs.json file. !!! tip Find the following contract addresses for the dynamic-{network}.yaml in the output files created at network launch: - zkevm.address-sequencer => createrollupoutput.json => sequencer - zkevm.address-zkevm => createrollupoutput.json => rollupAddress - zkevm.address-admin => deployoutput.json => admin - zkevm.address-rollup => deployoutput.json => polygonRollupManagerAddress - zkevm.address-ger-manager => deployoutput.json => polygonZkEVMGlobalExitRootAddress 4. Mount the directory containing the config files on a Docker container. For example /dynamic-mynetwork. 5. To use the new config when starting erigon, use the --config flag with the path to the config file e.g. --config="/dynamic-mynetwork/dynamic-mynetwork.yaml". # deploy-cdk-erigon.md: Prerequisites There are a few prerequisites for successfully deploying the CDK node. Hardware requirements Operating system: A Linux-based OS (e.g. Ubuntu 22.04 LTS). Memory: At least 64GB RAM. CPU: A minimum 4-core CPU, or higher (both Apple Silicon and amd64 are supported). !!! tip - On x86, the following packages are required to use the optimal, vectorized-Poseidon-hashing for the sparse Merkle tree: - Linux: libgtest-dev libomp-dev libgmp-dev - MacOS: brew install libomp brew install gmp - For Apple silicon, the iden3 library is used instead. Software The installation requires Go 1.19. Set up 1. Clone the repo and cd to the root: sh git clone https://github.com/0xPolygonHermez/cdk-erigon cd cdk-erigon/ 2. Install the relevant libraries for your architecture by running: sh make build-libs L1 interaction In order to retrieve data from L1, the L1 syncer needs to know how to request the highest block. This can be configured by the flag: zkevm.l1-highest-block-type. The flag defaults to retrieving the finalized block. However, there are cases where you may wish to pass safe or latest. Set up sequencer !!! warning "Work in progress" - Sequencer is production ready from v2.x.x onwards. - Please check the roadmap for more information. Enable the sequencer by setting the following environment variable: sh CDKERIGONSEQUENCER=1 ./build/bin/cdk-erigon Special mode - L1 recovery The sequencer supports a special recovery mode which allows it to continue the chain using data from the L1. To enable this, add the following flag: sh zkevm.l1-sync-start-block: [first l1 block with sequencer data]. !!! important Find the first block on the L1 from the sequencer contract that contains the sequenceBatches event. When the node starts up, it pulls the L1 data into the cdk-erigon database and uses it during execution, effectively rebuilding the chain from the L1 data rather than waiting for transactions from the transaction pool. You can use this in tandem with unwinding the chain or by using the zkevm.sync-limit flag to limit the chain to a specific block height before starting the L1 recovery. This is useful if you have an RPC node available to speed up the process. !!! warning If using the zkevm.sync-limit flag, you need to go to the boundary of a batch+1 block; so if batch 41 ends at block 99 then set the flag to 100. Enable zkEVM APIs In order to enable the zkevm namespace, add zkevm to the http.api flag. Supported functions - zkevmbatchNumber - zkevmbatchNumberByBlockNumber - zkevmconsolidatedBlockNumber - zkevmisBlockConsolidated - zkevmverifiedBatchNumber - zkevmisBlockVirtualized - zkevmvirtualBatchNumber - zkevmgetFullBlockByHash - zkevmgetFullBlockByNumber - zkevmvirtualCounters - zkevmtraceTransactionCounters Supported functions (remote) - zkevmgetBatchByNumber Not yet supported - zkevmgetNativeBlockHashesInRange Deprecated - zkevmgetBroadcastURI - removed by zkEVM. Warnings - The instantiation of Poseidon over the Goldilocks field is much faster on x86, but developers using MacOS M1/M2 chips may experience slower processing. - Developers should avoid falling significantly behind the network, especially for longer chains, as this triggers an SMT rebuild, which takes a considerable amount of time to complete. Configuration files - Config files are the easiest way to configure cdk-erigon. - There are examples files in the repository for each network; e.g. hermezconfig-mainnet.yaml.example. - Depending on your RPC provider, you may wish to alter zkevm.rpc-ratelimit in the yaml file. Running CDK Erigon 1. Build the node with the following command: sh make cdk-erigon 2. Set up your config file by copying one of the examples found in the repository root directory, and edit as required and add your network name to the following command. sh run ./build/bin/cdk-erigon --config="./hermezconfig-{network}.yaml" !!! warn Be aware that the --externalcl flag is removed upstream in cdk-erigon so take care when reusing commands/configurations. Run modes cdk-erigon can run as an RPC node which uses the data stream to fetch new block/batch information and track a remote sequencer. This is the default behavior. It can also run as a sequencer. To enable the sequencer, set the CDKERIGONSEQUENCER environment variable to 1 and start the node. !!! warning "Work in progress" - Sequencer is production ready from v2.x.x onwards. - Please check the roadmap for more information. cdk-erigon supports migrating a node from being an RPC node to a sequencer and vice versa. To do this, stop the node, set the CDKERIGONSEQUENCER environment variable to the desired value and restart the node. Please ensure that you do include the sequencer specific flags found below when running as a sequencer. You can include these flags when running as an RPC to keep a consistent configuration between the two run modes. Docker (DockerHub) The image comes with three preinstalled default configurations which you can edit according to the configuration section below; otherwise you can mount your own config to the container as necessary. A datadir must be mounted to the container to persist the chain data between runs. Example docker commands Mainnet sh docker run -d -p 8545:8545 -v ./cdk-erigon-data/:/home/erigon/.local/share/erigon hermeznetwork/cdk-erigon --config="./mainnet.yaml" --zkevm.l1-rpc-url=https://rpc.eth.gateway.fm Cardona sh docker run -d -p 8545:8545 -v ./cdk-erigon-data/:/home/erigon/.local/share/erigon hermeznetwork/cdk-erigon --config="./cardona.yaml" --zkevm.l1-rpc-url=https://rpc.sepolia.org Example docker-compose commands Mainnet sh NETWORK=mainnet L1RPCURL=https://rpc.eth.gateway.fm docker-compose -f docker-compose-example.yml up -d Cardona: sh NETWORK=cardona L1RPCURL=https://rpc.sepolia.org docker-compose -f docker-compose-example.yml up -d Configurations The following examples are comprehensive. There are key fields which must be set, such as datadir, and some you may wish to change to increase performance, such as zkevm.l1-rpc-url as the provided RPCs may have restrictive rate limits. - datadir: Path to your node's data directory. - chain: Specifies the L2 network to connect with; e.g. hermez-mainnet. For dynamic configs this should always be in the format dynamic-{network}. - http: Enables HTTP RPC server (set to true). - private.api.addr: Address for the private API, typically localhost:9091. Change this to run multiple instances on the same machine. - zkevm.l2-chain-id: Chain ID for the L2 network; e.g. 1101. - zkevm.l2-sequencer-rpc-url: URL for the L2 sequencer RPC. - zkevm.l2-datastreamer-url: URL for the L2 data streamer. - zkevm.l1-chain-id: Chain ID for the L1 network. - zkevm.l1-rpc-url: L1 Ethereum RPC URL. - zkevm.address-sequencer: The contract address for the sequencer. - zkevm.address-zkevm: The address for the zkevm contract. - zkevm.address-admin: The address for the admin contract. - zkevm.address-rollup: The address for the rollup contract. - zkevm.address-ger-manager: The address for the GER manager contract. - zkevm.rpc-ratelimit: Rate limit for RPC calls. - zkevm.data-stream-port: Port for the data stream. This needs to be set to enable the datastream server. - zkevm.data-stream-host: The host for the data stream i.e. localhost. This must be set to enable the datastream server. - zkevm.datastream-version: Version of the data stream protocol. - externalcl: External consensus layer flag. - http.api: List of enabled HTTP API modules. Sequencer specific config - zkevm.executor-urls: A csv list of the executor URLs. These are used in a round robbin fashion by the sequencer. - zkevm.executor-strict: Default is true but can be set to false when running the sequencer without verifications (use with extreme caution). - zkevm.witness-full: Default is true. Controls whether the full or partial witness is used with the executor. - zkevm.sequencer-initial-fork-id: The fork id to start the network with. Useful config entries - zkevm.sync-limit: This ensures the network only syncs to a given block height. # index.md: CDK Erigon CDK Erigon is the implementation of Erigon adapted to be a specialized framework for creating and managing chains that run the Polygon zkEVM protocol. Its code repository can be found at 0xPolygonHermez/cdk-erigon. Erigon, formerly known as Turbo-Geth, is a high-performance Ethereum client built to meet the increasing demands of the Ethereum blockchain. It focuses on optimizing performance, disk space, and synchronization speed. Since its inception, Erigon has demonstrated the ability to sync in full archive node mode without requiring advanced hardware or weeks of synchronization time. The CDK Erigon framework utilizes Erigon's fast syncing mode, allowing any CDK Erigon RPC node to sync with the CDK Erigon sequencer node. Polygon zkEVM components The legacy Polygon zkEVM, either as a rollup or validium, has the following components: - RPC node through which transactions are submitted. - Pool DB for storing users' transactions. - Sequencer node for executing transactions, creating blocks and batches, sending batches to L1 (or DAC), and sequencing them. - Aggregator node for facilitating proving and verification. - Synchronizer for keeping sync with L1. - State DB for permanently storing state data (but not the Merkle trees). For the sake of simplicity, we leave out the bridge and consensus smart contracts. CDK Erigon components CDK Erigon, an Erigon implementation running the Polygon zkEVM protocol, includes most of the above components with a few modifications to fully leverage Erigon's capabilities. CDK Erigon consists of the following components: - CDK Erigon RPC node through which transactions are submitted. - tx-pool-manager for storing users' transactions. - CDK Erigon sequencer node for executing transactions, and creating blocks and batches. - SequenceSender for sequencing batches. - In the case of a rollup, the SequenceSender sends batch data and the sequenceBatches transaction to L1. - In the case of a validium, the SequenceSender sends batch data to the Data Availability Committee (DAC), requests for signatures from the DAC, and sends the sequenceBatchesValidium transaction to L1. - Aggregator node for facilitating proving and verification. Transaction flow Here is a high-level overview of how user transactions move through the system, from the moment they are submitted via an RPC node to when they are finalized. This process involves transitioning through three states: trusted, virtual, and verified. The figure below depicts a simplified CDK Erigon architecture and the flow of transactions, specifically in the rollup mode. !Figure: Rollup tx flow Trusted state User's transactions are submitted via a CDK Erigon RPC node, and stored in the Pool DB. The CDK Erigon sequencer selects transactions from the Pool DB according to its own strategy. It executes the transactions, adds the successfully executed ones to blocks, and then groups the blocks into batches. At this stage the transactions have reached the trusted state, and the RPC node syncs with the sequencer. Virtual state The SequenceSender fetches batches from the CDK Erigon sequencer. The SequenceSender sends the batch data to either L1 or DAC for data availability, depending on whether the chain is in a rollup or validium mode. - If the chain is a validium, SequenceSender also requests for the required number of DAC-member signatures. As mentioned above, SequencerSender sends either a sequenceBatches or sequenceBatchesValidium transaction to L1. At this stage the transactions have reached the virtual state. Verified state The CDK Erigon RPC node checks if the batches received by syncing with the sequencer have been virtualized, and thus validate their correctness against the data on L1. The aggregator retrieves the witness from the CDK Erigon RPC and fetches data required for proving batches from L1. The witness and batch data are sent to the Prover as a request for a proof. The Prover sends back a verifiable proof, called batchProof. The aggregator sends the batchProof to the consensus smart contract on L1 for verification. Once verified, the transactions have now reached the verified state. The figure below depicts a simplified CDK Erigon architecture and the flow of transactions, in the validium mode. !Figure: Validium tx flow Bridge To maintain simplicity, the architectural diagram above omits the bridge and consensus smart contracts. CDK Erigon deploys the LxLy bridge, a pair of identical smart contracts on L1 and L2. It is called the 'LxLy bridge' to indicate that it manages not only exits between L1 and L2, but also between different L2s, for example. The bridge enables asset deposits on one layer and withdrawals on the destination network. When a deposit is made on L1, the bridge waits for verification of the corresponding Ethereum block before allowing the recipient on L2 to claim the asset. For a deposit on L2, the bridge waits for the related batch to be verified before unlocking the asset on L1 for the recipient to withdraw. Standard Erigon The standard Erigon design follows a modular architecture, consisting of a P2P network, transaction pool, consensus engine, Core with its State DB, API service, and JSON RPC interface. Although a snapshots component is included in the design, it is not yet under development. While standard Erigon is similar to other nodes like Geth, it is distinguished by its Core and State DB component. Components such as the P2P network and consensus engine are largely the same as in Geth. The figure below depicts a simplified, high-level design of Erigon's architecture. !Figure: Erigon architecture Storing Merkle trees State in Ethereum is represented in the form Merkle Patricia tries, and it consists of, - A tree of accounts. - A tree of storage slots. - A tree of receipts. Merkle trees provide a conceptual view of the state, but storing them in memory requires flattening the data. Typically, a Merkle tree is represented in storage as a mapping of each hash value to its child nodes. Example Letâ€™s look at an example of how a Merkle tree is represented in storage. Consider a binary Merkle tree, such as the sparse Merkle trees (SMTs) used in Polygon zkEVM. The figure below illustrates an SMT and its representation in storage. !Figure: SMT memory representation Flat DB storage Merkle trees offer an convenient mechanism for proving and verifying the presence of specific data within a state. However, retrieving a value from a specific leaf often requires traversing the full height of the Merkle tree, which is inefficient. Since Merkle proofs contain only a few intermediate hash values, which are sufficient for verification, it is unnecessary to store all intermediate hashes. Erigon, therefore, eliminates the merkelization of state data in storage and replaces it with a flat database structure. Data in Erigon State DB is stored as key-value pairs where, - The account address is the key pointing to the account data. For example, $$ \mathtt{0x0123 \dots E38F}\ \rightarrow\ \texttt{account data} $$ - The account address is concatenated with the storage slot address to form the key, while the storage slot itself serves as the value. For example, $$ \mathtt{0x0123 \dots E38F} || \mathtt{0xFF1D \dots B12D} \ \rightarrow\ \texttt{storage slot data} $$ These key-value pairs are stored in a table called Plain state. An illustration of three such pairs is shown below. !Figure: Erigon plain state For compatibility with Ethereum, a second table called Hash state is used to store hash digests of the values in the Plain state. While this results in storing the same amount of data twice, it still leads to a significant reduction in memory usage. It offers a 10:1 reduction ratio compared to the legacy Ethereum node. In other words, 300GB of data stored in a legacy Ethereum node is equivalent to 30GB in Erigonâ€™s flat DB storage. Another advantage of flat DB storage is the ease and speed of value lookups, compared to traversing the full height of a Merkle tree to retrieve a leaf value. Separate commitment Since flat DB storage no longer uses merkelization, how does Erigon enable proof and verification? Erigon separates commitment from data. In other words, while the flat DB storage is used for storing data, merkelization is employed for commitments. A commitment is a binary Merkle tree, similar to the SMT used in Polygon zkEVM, and is stored in a separate table within the database. Once the commitment tree is built, Erigon uses a cached version of the tree, called Intermediate Hashes, to track changes in state data. However, due to the absence of prior Intermediate Hashes, the initial execution of transactions requires building the commitment tree from scratch using data from the flat DB storage. Subsequent iterations then use the Intermediate Hashes cache. Although it is currently a rare occurrence, jumping ahead by many blocks to rebuild the commitment tree can be much faster than rebuilding it iteratively. RPC node The remote procedure call (RPC) interface in Erigon allows external applications to interact with the underlying blockchain. The interface is essential for decentralized applications such as dApps, wallets, and other blockchain services. The RPC provides methods for querying blockchain data, sending transactions, and managing accounts. The following sequence diagram shows how a transaction is processed by the Erigon node. !Figure: CDK Erigon RPC node Distinguishing features Erigon is characterized by the following features: Modular architecture, optimized performance, reduced disk usage, and fast synchronization. - Erigon has a modular architecture, which makes it highly efficient and customizable for various common blockchain tasks. This allows different components to be developed, optimized, and updated independently. Such a separation of concerns helps in improving the performance and reliability of each module. - Erigon employs advanced techniques for data handling, such as memory-mapped files and optimized data structures, to ensure high-speed processing of blockchain data. - By implementing a more efficient database schema, Erigon significantly reduces disk usage compared to other Ethereum clients. - Erigon's fast sync method allows nodes to catch up with the blockchain more quickly by downloading only the most recent state of the blockchain, rather than the entire history. CDK Erigon vs. Polygon zkEVM At a high level, CDK Erigon works similarly to Polygon zkEVM, but with a few differences. Like Polygon zkEVM, CDK Erigon has sequencer and RPC nodes, except that these are Erigon nodes implemented to serve as the sequencer and RPC, respectively. While the Polygon zkEVM sequencer creates batches and posts them to L1, the CDK Erigon sequencer does not perform these tasks. Instead, they are handled by the SequenceSender. In the validium mode, it is again the SequenceSender, not the Sequencer, that interacts with the Data Availability Committee (DAC). Data streamer As seen in the transaction flow above, State DB does not appear in the CDK Erigon architectural diagrams. This is because CDK Erigon currently shares the L2 state via the data streamer (DS) library. Data streamer was developed to serve raw block data to external nodes that need to maintain an up-to-date L2 state. Any CDK Erigon node can function as both a DS stream server and client, except for the CDK Erigon sequencer, which only has a DS stream server. Both the SequenceSender and RPC nodes have a DS stream server and client, allowing them to request batches from the CDK Erigon sequencer and serve batches to other nodes. For instance, the Aggregator has a DS stream client, enabling it to request batch data from the CDK Erigon RPC. The figure below illustrates the data stream servers and clients in CDK Erigon. !Figure: DS server client L1 syncing and recovery Another key difference between CDK Erigon and Polygon zkEVM is how they sync with L1. It consists of two parts, - Firstly, L1 syncing in normal operation mode. - Secondly, L1 recovery or DAC recovery is used, depending on whether the network is implemented in rollup or validium mode. L1 syncing Normal operation mode refers to both the CDK Erigon sequencer and RPC nodes reading all data related to sequences, verifications, and the information needed to build the L1InfoTree from L1 smart contracts. Each CDK Erigon sequencer or RPC node has its own fully built-in L1InfoTree, which can be queried via the RPC, similar to the Polygon zkEVM node. Sequences and verifications are primarily needed for GetBatchByNumber and other API methods like verifiedBatchNumber. All sequences are important because the L1InfoTree is used in block building, proving batches, and is essential for the proper functioning of the bridge. L1 recovery Although it uses the same mechanism as L1 syncing, L1 recovery is performed if data is lost in all instances of Erigon. Between any two CDK Erigon nodes, either can function as the sequencer while the other serves as the RPC, and these roles are interchangeable. Both nodes store roughly the same data, as each maintains a full L1InfoTree. If both instances of CDK Erigon are lost, including all backups, it is still possible to retrieve batch data from L1 (or the DAC) and rebuild the state. This process is called L1 recovery, and it is clearly not part of normal operation mode. CDK Erigon vs. standard Erigon Next is a quick look at how CDK Erigon is different from standard Erigon. - The first difference between CDK Erigon and standard Erigon is the integration of Polygon zkEVM consensus protocols. - CDK Erigon is an adaptation of standard Erigon, optimized specifically for L2 scaling solutions where high throughput and reduced transaction costs are of paramount importance. - Erigon's modular design is ideal for the CDK implementation, as it aligns with the underlying modular architecture of Polygon zkEVM. - CDK Erigon is designed to seamlessly integrate with the broader Polygon ecosystem, facilitating interoperability with Polygon's products, services, scaling solutions, and tools. - The RPC interface in the CDK implementation is adapted to support the functionalities and optimizations of the zkEVM protocol, enabling more efficient communication and interaction with the network. # network-recovery.md: CDK-erigon supports two simple methods for network recovery: - Partial L1 recovery; from a partially synced datadir. - Full L1 recovery; from a completely empty datadir. Partial recovery Sync limit step First, find out the following: - What batch to set as the first batch to start recovering from. - What is the last block of the batch prior to this. It can be found by querying the RPC or checking the L1 data from the sequencer contract. Once we know the last block number, we can begin a fresh sync to get to that block height. To do this use the same configuration that you would normally use for the network but add an additional flag in --zkevm.sync-limit=[block you need + 1]. !!! info "Example" In order to sync to block 100, enter 101 for the flag value. Let the node run and it eventually sits in a loop at this block height. Once the node reaches the required height, you can stop the node as normal. L1 recovery step First determine the earliest L1 block height suitable for recovery. You can do this by looking for the L1 block number for the earliest transaction against the sequencer contract (found in the cdk-erigon config). Once you have the info you need, start the node up with a new flag: zkevm.l1-sync-start-block=[l1block height]. !!! important Remove the zkevm.sync-limit flag from the previous step at this point if you are running a partial recovery. It is important to pick the earliest block on the network so that the L1 info tree update events are gathered correctly. If not, you run the risk of the indexes not lining up. Full recovery Follow the L1 recovery step as above, and use a completely fresh datadir. # releases.md: cdk-erigon is a fork of erigon and is currently in alpha. It is optimized for syncing with the Polygon zkEVM network. Current chain/fork support status At the time of writing, cdk-erigon supports the following chains and fork ids: - zkEVM Cardona testnet: full support. - zkEVM mainnet: beta support. - CDK chains: beta support (forkid.9 and above). Roadmap - v1.1.x: RPC (full support). - v2.x.x: Sequencer (full support). - v3.x.x: Erigon 3 based (snapshot support). # resources.md: Networks | Network Name | Chain ID | ForkID | Genesis File | RPC URL | Rootchain | Rollup Address | |------------------|----------|--------|--------------|--|------------------|--| | zkEVM Mainnet | 1101 | 9 | Link | Mainnet RPC | Ethereum Mainnet | 0x5132A183E9F3CB7C848b0AAC5Ae0c4f0491B7aB2 | | zkEVM Cardona | 2442 | 9 | Link | Cardona RPC | Sepolia | 0x32d33D5137a7cFFb54c5Bf8371172bcEc5f310ff | Block explorers - Mainnet: PolygonScan mainnet - Cardona: PolygonScan Cardona # index.md: This glossary provides definitions for technical terminology and concepts that commonly occur throughout the CDK docs space. AggLayer v1 (AL1) The AggLayer is the interoperability layer that CDK chains connect to, enabling features such as seamless and efficient cross-chain communication, unified liquidity, and more. AggLayer v1 (AL1), is the first version of many planned iterations that relies on ZK checks to ensure operational soundness, and a unified bridge infrastructure. Read more on the AggLayer in the Polygon blog here. Chain operator The IP, or individual(s), who own a chain and are responsible for chain operation and maintenance. This includes tasks such as transaction validation, block production, and ensuring the security and integrity of the chain, etc., any combination of which a chain operator may perform personally in varying degrees. Data availability Data availability in the context of modular rollups refers to the idea that transaction callback data needs to be available to L1 network actors where transactions are settled and finalized, so it can be used to verify transaction execution if necessary. Data Availability Committee (DAC) Polygon CDK validiums connect to a DAC to guarantee data availability. The DAC nodes fetch tx data from the sequencer, validate it independently, and then sign it guaranteeing its validity before storing it in the local database. The data remains available to be fetched by other networks actors across all DAC nodes. LxLy bridge The native bridge infrastructure of CDK chains that allows transfer of assets and messages between L2 and L1 (typically Ethereum). LxLy messenger A contract on the LxLy bridge utilizes its message passing capabilities to pass arbitrary messages between L1 and L2. This is not a separate component, but part of the LxLy bridge's architecture. POL (Token) The POL token powers the Polygon ecosystem through a native re-staking protocol that allows token holders to validate multiple chains, and perform multiple roles on each of those chains (sequencing, ZK proof generation, participation in data availability committees, etc.) Rollups Rollups refer to blockchain scaling solutions (in the context of Ethereum) that carry out transaction execution on L2, and then post updated state data to a contract on L1. There are different types rollups, two of the most popular being optimistic, and zero-knowledge (ZK) rollups. Follow the links below to learn more: - Optimistic rollups - ZK rollups Stake the Bridge (STB) A feature of the unified bridge that lets CDK chain operators maintain control over the assets that are deposited to their respective networks, enabling them to implement staking mechanisms, investment strategies, and other custom features on L2. Unified bridge A specific instance of an LxLy bridge that allows several chains to connect to it. This instance is specific to the AggLayer v1. Unified escrow (Master Escrow) The unified bridge's escrow contract that holds all the tokens that bridged (locked on L1), and natively minted on L2. Validiums Validiums are a special kind of ZK rollup protocol that handles data availability off-chain instead of posting callback data to base layer Ethereum. CDK chains support deploying and running validium using a Data Availability Committee (DAC) as the DA solution. Learn more about validiums here. # connect-testnet.md: Stavanger !!! warning - Stavanger is waiting for a stable redeployment. - You may experience errors on the testnet until the next CDK release. The CDK Stavanger testnet is a validium testnet based on Sepolia. - Add the RPC network details to your wallet by navigating to the add network input and entering the data as given in the table below. - Use the faucet to get test ETH. - Bridge assets from Sepolia to Stavanger using the bridge. - Confirm your transactions with the block explorer. !!! tips "Setting up your wallet" - Click Connect wallet on the Stavanger explorer to auto set up your wallet. - Check out the latest on setting up a custom network with MetaMask. Stavanger network details | Name | Value | Usage | | ------- | ----------- | --------- | | JSON RPC | https://sn2-stavanger-rpc.eu-north-2.gateway.fm | Make remote procedure calls to the CDK testnet. | | Faucet | https://sn2-stavanger-faucet.eu-north-2.gateway.fm | Get testnet ETH | | Bridge | https://sn2-stavanger-bridge.eu-north-2.gateway.fm | Bridge tokens from Sepolia | | Block explorer | https://sn2-stavanger-blockscout.eu-north-2.gateway.fm | Confirm transactions with the explorer | | Chain id | 686669576 | Chain identification value | | Currency | ETH | Test token | Blackberry The Blackberry testnet is a zero-knowledge-powered, layer 2 testnet based on Sepolia. It uses Polygon CDK for transaction validity while keeping transaction data off-chain using DACs. - Add the RPC network details to your wallet by navigating to the add network input and entering the data as given in the table below. - Obtain Sepolia ETH from the public faucets available. - Bridge assets from the Ethereum Sepolia network to Blackberry using the bridge service. - Confirm your transactions with the block explorer. !!! tips "More information" - For more information on bridging and faucet services on Blackberry, head over to the Gelato documentation. Blackberry network details | Name | Value | Usage | | -------------- | -- | --- | | JSON RPC | https://rpc.polygon-blackberry.gelato.digital | Make remote procedure calls to the Blackberry testnet. | | Faucet | https://www.alchemy.com/faucets/ethereum-sepolia | Obtain sepolia ETH | | Bridge | https://bridge.gelato.network/bridge/polygon-blackberry | Bridge assets from the ethereum sepolia network | | Block explorer | https://polygon-blackberry.gelatoscout.com/ | Confirm transactions with the explorer | | Chain id | 94204209 | Chain identification value | | Currency | sETH | Test token | # deploy-t1-prover-devnet.md: This document shows you how to deploy a docker-compose file for running a fully-functional, local development network (devnet) for Ethereum with proof-of-stake enabled. The configuration uses Prysm as a consensus client, with either geth or erigon as an execution client. The setup is a single node devnet with 64 deterministically-generated validators[^1] to drive the creation of blocks in an Ethereum proof-of-stake chain. The devnet is fully functional and allows for deployment of smart contracts and all features that come with the Prysm consensus client, such as its rich set of APIs for retrieving data from the blockchain. Running a devnet like this provides the best way to understand Ethereum proof-of-stake under the hood, and gives allowance for devs to tinker with various settings that suit their system design. Running the devnet 1. Checkout this repository and install docker. 2. Run the following command to fire up the devnet containers: bash docker compose up -d You should see the statuses of the containers as shown below: example $ docker compose up -d [+] Running 7/7 [+] Running 10/10 âœ” Container eth-pos-devnet-create-beacon-chain-genesis-1 Exited âœ” Container eth-pos-devnet-create-beacon-node-keys-1 Exited âœ” Container eth-pos-devnet-beacon-chain-2-1 Started âœ” Container eth-pos-devnet-beacon-chain-1-1 Started âœ” Container eth-pos-devnet-geth-genesis-1 Exited âœ” Container eth-pos-devnet-geth-import-1 Exited âœ” Container eth-pos-devnet-erigon-genesis-1 Started âœ” Container eth-pos-devnet-validator-1 Started âœ” Container eth-pos-devnet-erigon-1 Started âœ” Container eth-pos-devnet-geth-1 Started 3. Stop the containers with this command: docker compose stop. 4. Before each restart, wiping old data with make clean. 5. Inspect the logs of launched services with this command: bash docker logs eth-pos-devnet-geth-1 -f Available features - Starts from the Capella Ethereum hard fork. - The network launches with a Validator Deposit Contract deployed at the address 0x4242424242424242424242424242424242424242. This can be used to onboard new validators into the network by depositing 32 ETH into the contract. - The default account used in the go-ethereum node is at the address 0x85da99c8a7c2c95964c8efd687e95e632fc533d6, which comes seeded with ETH for use in the network. This can be used to send transactions, deploy contracts, and more. - The default account at 0x85da99c8a7c2c95964c8efd687e95e632fc533d6 is also set as the fee recipient for transaction fees proposed by validators in Prysm. This address will be receiving the fees of all proposer activity. - The go-ethereum JSON-RPC API is available at http://geth:8545. - The Prysm client's REST APIs are available at http://beacon-chain:3500. For more info on what these APIs are, see here - The Prysm client also exposes a gRPC API at http://beacon-chain:4000. Type 1 prover testing procedure The aim of this devnet setup is to use Polygon Type 1 Prover to test Erigon state witnesses. The following steps create some test data. 1. Start the devnet up with docker compose up. If you had previously run this command, you might want to wipe old data with a make clean to avoid running from a previous state. 2. Wait for blocks to start being produced. This should only take a few seconds. You can use polycli monitor to quickly check that blocks are being created. 3. Generate some load and test transactions, by using a tool like polycli to create transactions. 4. Once the load is done, and if you ran docker in detached mode, you can stop the devnet with docker compose stop. 5. Checkout and build jerrigon from the feat/zero branch. You can use make all to build everything. 6. Create a copy of the erigon state directory to avoid corrupting things bash sudo cp -r execution/erigon/ execution/erigon.bak sudo chown -R $USER:$USER execution/erigon.bak/ 7. Now we can start the Jerrigon fork of Erigon. This will give us RPC access to the state that we created in the previous steps. bash /code/jerrigon/build/bin/erigon \ --http \ --http.api=eth,net,web3,erigon,engine,debug \ --http.addr=0.0.0.0 \ --http.corsdomain= \ --http.vhosts any \ --ws \ --nodiscover=true \ --txpool.disable=true \ --no-downloader=true \ --maxpeers 0 \ --datadir=./execution/erigon.bak 8. With the RPC running we can retrieve the blocks, witnesses, and use zero-bin to parse them. In one particular test case below, about 240 blocks worth of data were generated. So, seq 0 240 was used for generating ranges of block numbers for testing purposes. bash Create a directory for storing the outputs mkdir out Call the zeroTracer to get the traces seq 0 240 | awk '{print "curl -o " sprintf("out/wit%02d", $0) ".json -H "'"'"Content-Type: application/json"'"'" -d "'"'"{\"method\":\"debugtraceBlockByNumber\",\"params\":[\"" sprintf("0x%X", $0) "\", {\"tracer\": \"zeroTracer\"}],\"id\":1,\"jsonrpc\":\"2.0\"}"'"'" http://127.0.0.1:8545"}' | bash download the blocks (this assumes you have foundry/cast installed) seq 0 240 | awk '{print "cast block --full -j " $0 " > out/block" sprintf("%02d", $0) ".json"}' | bash 9. At this point, we'll want to checkout and build zero-bin in order to test proof generation. Make sure to checkout that repo and run cargo build --release to compile the application for testing. The snippets below assume zero-bin has been checked out and compiled in $HOME/code/zero-bin. After compiling, the leader and rpc binaries will be created in the target/release folder. bash use zero-bin to convert witness formats. This is a basic test seq 0 240 | awk '{print "/code/zero-bin/target/release/rpc fetch --rpc-url http://127.0.0.1:8545 --block-number " $0 " > " sprintf("out/zero%02d", $0) ".json" }' | bash use zero-bin to generate a proof for the genesis block ./leader --arithmetic 16..23 --byte-packing 9..21 --cpu 12..25 --keccak 14..20 --keccak-sponge 9..15 --logic 12..18 --memory 17..28 --runtime in-memory -n 1 jerigon --rpc-url http://127.0.0.1:8545 --block-number 1 --proof-output-path 1.json seq 2 240 | awk '{print "./leader --arithmetic 16..23 --byte-packing 9..21 --cpu 12..25 --keccak 14..20 --keccak-sponge 9..15 --logic 12..18 --memory 17..28 --runtime in-memory -n 4 jerigon --rpc-url http://127.0.0.1:8545 --block-number " $1 " --proof-output-path " $1 ".json --previous-proof " ($1 - 1) ".json"}' Operational notes - Pay attention to memory usage on the system running zero-bin. Certain transactions can consume a lot of memory and lead to an out-of-memory (OOM) error. - You'll want to run zero-bin on a system with at least 32GB of RAM. - When you run zero-bin, a local file will be created with a name like proverstate. This file needs to be deleted if any of the circuit sizes are changed. - There is a useful script in zero-bin to run a range of proofs. !!! important Both the state witness generation and decoding logic are actively being improved. We expect that the following transaction types or use-cases to prove without any issues: - Empty blocks (important use case) - EOA transfers - ERC-20 mints & transfers - ERC-721 mintes & transfers Shortcuts 1. There is a shortcut that creates the genesis file allocations for our mnemonic which has already been hard-coded into the genesis file. However, if you want to use a different testing account, use the one below. bash polycli wallet inspect --mnemonic "code code code code code code code code code code code quality" | jq '.Addresses[] | {"key": .ETHAddress, "value": { "balance": "0x21e19e0c9bab2400000"}}' | jq -s 'fromentries' [^1]: See Line11 of the docker-compose.yml https://github.com/0xPolygonZero/eth-pos-devnet-provable/blob/959da56673c25c2094b1a23bc9e1fa9ae9a9db6e/docker-compose.yml#L11 # deploy-t1-prover.md: This document shows you how to run the Polygon Type 1 Prover, specifically for proving transactions, but with the option to test full blocks of less than 4M gas, which means it is similar to eth-proof but for transaction proofs. Quick start There are two ways to run the prover. The simplest way to get started is to use the in-memory runtime of Paladin. This requires very little setup, but it's not really suitable for large scale testing. The other method for testing the prover is to use an AMQP like RabbitMQ to distribute the workload over many workers. !!! info It's worth noting that you'll need at least 40GB of physical memory to run the prover. Setup Start by cloning the repo here. Before running the prover, compile the application. bash env RUSTFLAGS='-C target-cpu=native' cargo build --release You should end up with two binaries in your target/release folder. One is called worker and the other is leader. Typically, we'll install these somewhere in our $PATH for convenience. Once you have the application available, you'll need to create a block witness which essentially serves as the input for the prover. Assuming you've deployed the leader binary, you should be able to generate a witness like this: bash paladin-leader rpc -u $RPCURL -t 0x2f0faea6778845b02f9faf84e7e911ef12c287ce7deb924c5925f3626c77906e > 0x2f0faea6778845b02f9faf84e7e911ef12c287ce7deb924c5925f3626c77906e.json You'll need access to an Ethereum RPC in order to run the command. The input argument is a transaction hash and in particular it is the last transaction hash in the block. Once you've successfully generated a witness, you're ready to start proving either with the in-memory runtime or the amqp runtime. In-memory proving Running the prover with the in-memory setup requires no setup. You can attempt to generate a proof with a command like this: bash env RUSTMINSTACK=33554432 \ ARITHMETICCIRCUITSIZE="15..28" \ BYTEPACKINGCIRCUITSIZE="9..28" \ CPUCIRCUITSIZE="12..28" \ KECCAKCIRCUITSIZE="14..28" \ KECCAKSPONGECIRCUITSIZE="9..28" \ LOGICCIRCUITSIZE="12..28" \ MEMORYCIRCUITSIZE="17..30" \ paladin-leader prove \ --runtime in-memory \ --num-workers 1 \ --input-witness 0x2f0faea6778845b02f9faf84e7e911ef12c287ce7deb924c5925f3626c77906e.json The circuit parameters here are meant to be compatible with virtually all Ethereum blocks. This creates a block proof from an input state root of the preceding block. You can adjust the --num-workers flag based on the number of available compute resources. !!! info "Rule of thumb" You probably want at least 8 cores per worker. AMQP proving Proving in a distributed compute environment depends on an AMQP server. We're not going to cover the setup of RabbitMQ, but assuming you have something like that available you can run a "leader" which distributes proving tasks to a collection of "workers" which actually do the proving work. In order to run the workers, use a command like: bash env RUSTMINSTACK=33554432 \ ARITHMETICCIRCUITSIZE="15..28" \ BYTEPACKINGCIRCUITSIZE="9..28" \ CPUCIRCUITSIZE="12..28" \ KECCAKCIRCUITSIZE="14..28" \ KECCAKSPONGECIRCUITSIZE="9..28" \ LOGICCIRCUITSIZE="12..28" \ MEMORYCIRCUITSIZE="17..30" \ paladin-worker --runtime amqp --amqp-uri=amqp://localhost:5672 This starts the worker and has it await tasks. Depending on your machine's system capacity, you can run several workers on the same operating system. An example systemd service is included. Once that service is installed, you can enable up to 16 workers on the same VM like this: bash seq 0 15 | xargs -I xxx systemctl enable paladin-worker@xxx seq 0 15 | xargs -I xxx systemctl start paladin-worker@xxx Now that you have your pool of paladin workers, you can start proving with a command like this: bash paladin-leader prove \ --runtime amqp \ --amqp-uri=amqp://localhost:5672 \ --input-witness 0x2f0faea6778845b02f9faf84e7e911ef12c287ce7deb924c5925f3626c77906e.json This command runs the same way as the in-memory mode except that the leader itself isn't doing the work. The separate worker processes are doing the heavy lifting. # integrate-da.md: This document shows you how to integrate a third-party data availability (DA) solution into your CDK stack. Prerequisites !!! tip Make sure you have upgraded your CDK stack if necessary. Set up contracts This section shows you how to create a custom CDK validium DAC contract. 1. Clone zkevm-contracts. 2. cd into zkevm-contracts and checkout tag v8.0.0-rc.3-fork.12. 3. Run npm install from the root. 4. cd to the contracts/v2/consensus/validium directory. !!! tip - Until further notice, these contracts run on the banana release. 5. Create your custom contract in the same directory, and make sure it implements the IDataAvailabilityProtocol interface. !!! tip - Use the Polygon DAC implementation contract: PolygonDataCommittee.sol as a guide. - The contract supports custom smart contract implementation and, through this, DACs can add their custom on-chain verification logic. 6. You can leave the verifyMessage function empty but make sure the getProcotolName function returns a unique name (such as Avail, Celestia, etc). The following example code comes from the PolygonDataCommitee.sol implementation. solidity // Name of the data availability protocol string internal constant PROTOCOLNAME = ""; ... / @notice Return the protocol name / function getProcotolName() external pure override returns (string memory) { return PROTOCOLNAME; } 7. Update the /deployment/v2/4createRollup.ts script to add your contract name. ts const supporteDataAvailabilityProtocols = [""]; 8. Make your contract deployable by copying, editing for your custom implementation, and pasting back in, the if statement from the /deployment/v2/4createRollup.ts#L251 node creation script. !!! info "PolygonValidiumEtrog.sol solution" The Etrog DAC integration contract is still work-in-progress at the time of writing but there are some interesting things to note. 1. It implements the function verifyMessage function: solidity // Validate that the data availability protocol accepts the dataAvailabilityMessage // note This is a view function, so there's not much risk even if this contract was vulnerable to reentrant attacks dataAvailabilityProtocol.verifyMessage(accumulatedNonForcedTransactionsHash, dataAvailabilityMessage); where accumulatedNonForcedTransactionsHash is used for verification against the protocol and dataAvailabilityMessage is a byte array containing the signature and addresses of the committee in ascending order. 2. It also implements a function to set the data availability protocol at line 287 to see how they do this. solidity / @notice Allow the admin to set a new data availability protocol @param newDataAvailabilityProtocol Address of the new data availability protocol / function setDataAvailabilityProtocol(IDataAvailabilityProtocol newDataAvailabilityProtocol) external onlyAdmin { dataAvailabilityProtocol = newDataAvailabilityProtocol; emit SetDataAvailabilityProtocol(address(newDataAvailabilityProtocol)); } Deploy Docker image This section shows you how to deploy the Docker image containing your custom DAC contract. 1. Edit the following parameters in the docker/scripts/v2/deployparametersdocker.json file: json "minDelayTimelock": 3600, // BECOMES "minDelayTimelock": 1, 2. Edit the following parameters in the /docker/scripts/v2/createrollupparametersdocker.json file: json "consensusContract": "PolygonValidiumEtrog", // CHANGE THIS TO YOUR CONTRACT NAME "dataAvailabilityProtocol": "PolygonDataCommittee", // ADD THIS PARAMETER 3. Run the following command: sh cp docker/scripts/v2/hardhat.example.paris hardhat.config.ts 4. Edit docker/scripts/v2/deploy-docker.sh to add the following line: sh sudo chmod -R go+rxw docker/gethData before docker build -t hermeznetwork/geth-zkevm-contracts -f docker/Dockerfile . 5. In the deployment/v2/4createRollup.ts file, uncomment the 290-291, and add a console.log output that grabs the address of the DAC: ts // Setup data commitee to 0 await (await polygonDataCommittee?.setupCommittee(0, [], "0x")).wait(); console.log(dataAvailabilityProtocol, "deployed to:", polygonDataCommittee.target); 6. Build the image with the following commands: sh sudo npx hardhat compile sudo npm run docker:contracts 7. Tag the image with the following command, where XXXX is custom: sh docker image tag hermeznetwork/geth-zkevm-contracts hermeznetwork/geth-cdk-validium-contracts:XXXX Set up the node This section shows you how to set up your CDK node that sends and receives data from the DAC. 1. Create a package that implements the DABackender interface and place it under the cdk-validium-node/tree/develop/dataavailability directory. 2. Add a new constant to the /dataavailability/config.go file that represents the DAC. go const (// DataAvailabilityCommittee is the DAC protocol backend DataAvailabilityCommittee DABackendType = "DataAvailabilityCommittee") where DataAvailabilityCommittee matches the PROTOCOLNAME see in the Set up contracts section. 3. OPTIONAL: Add a config struct to the new package inside the main config.go file so that your package can receive custom configurations using the nodeâ€™s main config file. 4. Instantiate your package and use it to create the main data availability instance, as done in the Polygon implementation. Test the integration !!! tip - By default, all E2E tests run using the DAC. - It is

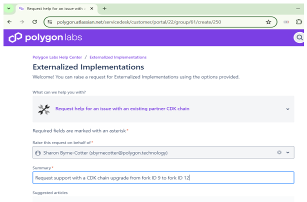
possible to run the E2E tests using other DAC backends by amending the test.node.config.toml file. To test your DAC integration, follow the steps below. 1. Create an E2E test that uses your protocol by following the test/e2e/datacommitteetest.go example. 2. Generate a docker image containing the changes to the node: sh make build-docker 3. Build the genesis file for the node: - First, clone the cdk-validium-node repo and checkout v0.6.4-cdk.5. - Edit the test/config/test.genesis.config.json file taking values in the generated output files created previously in the contract repo's docker/deploymentOutputs folder: !!! info "Parameters to change" l1Config.polygonZkEVMAddress ==> rollupAddress @ createrollupoutput.json l1Config.polygonRollupManagerAddress ==> polygonRollupManager @ deployoutput.json l1Config.polTokenAddress ==> polTokenAddress @ deployoutput.json l1Config.polygonZkEVMGlobalExitRootAddress ==> polygonZkEVMGlobalExitRootAddress @ deployoutput.json rollupCreationBlockNumber ==> createRollupBlock @ createrollupoutput.json rollupManagerCreationBlockNumber ==> deploymentBlockNumber @ deployoutput.json root ==> root @ genesis.json genesis ==> genesis @ genesis.json !!! important - You should follow this step every time you build a new Docker image. 4. Update the contracts Docker image tag with the custom tag you created at the deploy Docker image section, step 7, by amending the node's Docker compose file. 5. Modify the Makefile so it can run your test. Use the Polygon DAC Makefile as an example. # manage-policies.md: Manage allowlists, and more, with policies !!! important Policies are currently only available in validium mode. Managing allowlists, denylists, and ACLs is done with policies. Policy overview A policy is a set of rules that govern what actions are allowed or denied in the transaction pool. - Fine-grained control: Developers can specify policies at a granular level, allowing or denying specific actions for specific addresses. - Dynamic updates: Policies and ACLs can be updated on-the-fly without requiring a node restart. - Database-backed: All policy data is stored in a PostgreSQL database. - Extensible: New policies can be easily added to the system. Validium node Policies Currently, there are two defined policies: - SendTx: governs whether an address may send transactions to the pool. - Deploy: governs whether an address may deploy a contract. The CDK validium node offers policy management features that include allowlisting[^1], denylisting[^2], and access control lists (ACLs)[^3]. These features are beneficial for validium-based app-chains that require fine-grained control over transaction pools. Code definitions - Policy management: cmd/policy.go contains the core logic of policy management. - Policy definitions: pool/policy.go contains structs and utility functions for policies and ACLs. - Data: pgpoolstorage/policy.go interacts with the data layer (PostgreSQL database) to store and retrieve policy and ACL data. - Policy interface: pool/interfaces.go contains a policy interface which defines the methods that policies must implement. How to use a policy | Command name | Description | Flags & parameters | |--------------|---|--| | policy add | Add address(es) to a policy exclusion list | --policy (or -p): Policy name
--csv: CSV file with addresses | | policy remove | Remove address(es) from a policy exclusion list | --policy (or -p): Policy name
--csv: CSV file with addresses to remove | | policy clear | Clear all addresses from a policy's exclusion list | --policy (or -p): Policy name | | policy describe | Describe the default actions for the policies or a specific policy | --policy (or -p): Policy name (optional)
--no-header: Omit header in output (optional) | | policy update | Update the default action for a policy | --policy (or -p): Policy name
--allow: Set policy to 'allow'
--deny: Set policy to 'deny' | !!! note The examples demonstrate a deploy policy. Add addresses To add one or more addresses to a specific policy, you can use the policy add command. If you have a CSV file containing the addresses, you can use the --csv flag. bash docker exec -it cdk-validium-aggregator /app/cdk-validium-node policy add --policy deploy 0xAddress1 Remove addresses To remove addresses from a policy, you can use the policy remove command. bash Remove a single address from the 'deploy' policy docker exec -it cdk-validium-aggregator /app/cdk-validium-node policy remove --policy deploy 0xAddress1 Remove multiple addresses from the 'deploy' policy using a CSV file docker exec -it cdk-validium-aggregator /app/cdk-validium-node policy remove --policy deploy --csv addresses.csv Clear all addresses To remove all addresses from a policy's ACL, you can use the policy clear command. bash docker exec -it cdk-validium-aggregator /app/cdk-validium-node policy clear --policy deploy Get information about a policy To get information about a specific policy or all policies, you can use the policy describe command. bash Describe a specific policy docker exec -it cdk-validium-aggregator /app/cdk-validium-node policy describe --policy deploy Describe all policies docker exec -it cdk-validium-aggregator /app/cdk-validium-node policy describe [^1]: Allowlisting: The process of explicitly allowing addresses to perform certain actions. [^2]: Denylisting: The process of explicitly denying addresses from performing certain actions. [^3]: ACL (access control list): A list of addresses that are exceptions to a given policy. # quick-test-stack.md: A quick and easy method for testing a running CDK stack, whether in validium or rollup mode, is by sending a zero-value transaction and examining the result. Get the URL First inspect the logs to find the zkevm-node-rpc details. For example, in the Kurtosis stack you should see something like: sh zkevm-node-rpc-001 http-rpc: 8123/tcp -> http://127.0.0.1:32803 Send transaction with cast In a terminal window, run the following command where the mnemonic is used just for testing and the address can be any valid account address. sh cast send --legacy --mnemonic 'code code code code code code code code code code code quality' --value 0 --gas-price 0 --rpc-url http://127.0.0.1:8123 0x0bb7AA0b4FdC2D2862c088424260e99ed6299148 You should see something like this as output: sh blockHash 0x5d6d45f46e54c5d0890dd8a4ede989dc8042d7d3aeada375ea11d2e77c91a298 blockNumber 1 contractAddress cumulativeGasUsed 21000 effectiveGasPrice 0 from 0x85dA99c8a7C2C95964c8EfD687E95E632Fc533D6 gasUsed 21000 logs [] logsBloom
0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
root 0x97b15537641583db08f1e3db15cb1e89212ed8d147670a11f93f368d5960e72f status 1 transactionHash 0xd5443cff8dcc1147ead09d978d3abe9179615aa3eecbe4819c6768390bc467a3 transactionIndex 0 type 0 to 0x66ecâ€¦89fd Status 1 signifies a successful transaction. Send transaction with MetaMask 1. Add the network details manually to MetaMask and accept the defaults and auto-populating fields. !MetaMask manual network entry 2. In local/test setup you will find a pre-funded account with private key: 0x12d7de8621a77640c9241b2595ba78ce443d05e94090365ab3bb5e19df82c625. Import the account into MetaMask.

!Import pre-funded account{ width=45% }

3. Send a transaction to another MetaMask account. !Transaction sending success
use-native-token.md: This document shows you how to set up and test a native token in a CDK stack. Summary !!! quote "From the Unified bridge documentation" When a token is utilized to cover gas expenses within a layer, we refer to it as the gas token for that specific layer. Even when employing a gas token within a layer, it remains feasible to transfer L1 ETH to that layer. In such instances, the ETH is recorded within an ERC20 contract known as W-ETH, functioning as another instance of an ERC20 token. The diagram below illustrates the interchange of assets between layers, focusing on LY as a layer of interest. It depicts several scenarios, such as bridging an ERC20 token from mainnet to another ERC20 token in LY, bridging L1 ETH to the LY gas token, or bridging a wrapped ERC20 token living on LX to LY ETH. !Unfied bridge exchanges Set up and prerequisites 1. Edit the params.yml file to set zkevmusegastokencontract to true. This forces a couple of changes at setup time: 1. During contract deployment, an ERC20 token is deployed on L1. 2. The rollup creation parameters file is modified to set the gasTokenAddress to the address of the L1 ERC20. 2. You can now run the stack: sh kurtosis run --enclave cdk-v1 --args-file params.yml --image-download always . It takes a few minutes to compile and deploy the full set of contracts. The screenshot below shows the full set of deployed services and highlights the bridge UI, L1 RPC, and L2 RPC services which we will focus on throughout this document. !Deployed services Mint gas token on L1 !!! tip For this section you will need to have Foundry, jq, and yq installed. 1. Extract the gasTokenAddress from the generated files: sh kurtosis service exec cdk-v1 contracts-001 "cat /opt/zkevm/createrollupparameters.json" 2. Run the following command to mint some L1 tokens. sh export gta=$(kurtosis service exec cdk-v1 contracts-001 "cat /opt/zkevm/createrollupparameters.json" | tail -n +2 | jq -r .gasTokenAddress) export l1rpcurl=$(kurtosis port print cdk-v1 el-1-geth-lighthouse rpc) cast send \ --mnemonic "code code code code code code code code code code code quality" \ --rpc-url "$l1rpcurl" \ "$gta" \ "mint(address,uint256)" \ 0x85dA99c8a7C2C95964c8EfD687E95E632Fc533D6 \ 1000000000000000000000000 3. Check the balance to make sure it worked. sh cast call --rpc-url "$l1rpcurl" "$gta" "balanceOf(address)(uint256)" 0x85dA99c8a7C2C95964c8EfD687E95E632Fc533D6 Open the bridge UI Run the following command to get the bridge UI URL and then open the URL in your browser. sh kurtosis port print cdk-v1 zkevm-bridge-proxy-001 web-ui Add L1 and L2 RPCs to your wallet As the URLs use HTTP, instead of HTTPS, you need to manually add them to MetaMask. 1. Retrieve the L1 RPC config by running the following command: sh echo \{\"networkname\": \"kurtosiscdkl1\", \"newrpcurl\": \"$l1rpcurl\", \"chainid\": $(yq .args.l1chainid params.yml), \"currencysymbol\": \"ETH\"\} | jq 2. Do the same for L2: sh echo \{\"networkname\": \"kurtosiscdkrollup\", \"newrpcurl\": \"$(kurtosis port print cdk-v1 zkevm-node-rpc-001 http-rpc)\", \"chainid\": $(yq .args.zkevmrollupchainid params.yml), \"currencysymbol\": \"CDK\"\} | jq Import an account If you used the pre-allocated mnemonic, you need to import an account using a private key. The first derived private key from the code...quality mnemonic is 42b6e34dc21598a807dc19d7784c71b2a7a01f6480dc6f58258f78e539f1a1fa. Bridging tokens Add token 1. On the bridge UI, click the token dropdown menu: !Token dropdown menu 2. Get the ERC20 gas token address from earlier and paste it in with: sh echo "$gta" In my case, the gas token address is 0xBDF337Ae0209B33285034c476f35733BFC890707. !Paste token in 3. Confirm the details and click the Add CDK Gas Token button. !Paste token in Bridge tokens 1. Select the token for bridging. !Select token for bridging 2. Add a value to bridge and click Continue. !Bridging details 3. Your bridge deposit should be pending. !Token bridging processing 4. After some time, the transaction should complete. !Bridging complete !!! tip If that doesn't happen, it's possible the auto-claiming service on L2 is not funded. Send it some Ether on L2 by running a command like this: sh cast send --legacy \ --value 10ether \ --private-key 0x12d7de8621a77640c9241b2595ba78ce443d05e94090365ab3bb5e19df82c625 \ --rpc-url "$(kurtosis port print cdk-v1 zkevm-node-rpc-001 http-rpc)" \ "$(yq -r .args.zkevml2claimtxmanageraddress params.yml)" 5. Switch to your L2 network on Metamask to see the bridged value on L2. !Check bridge success on L2 in MetaMask Transfer token 1. To complete the test, lets transfer some of the value on L2 to another address. !Transfer to another L2 address 2. Check the balance in MetaMask after importing the private key 0903a9a721167e2abaa0a33553cbeb209dc9300d28e4e4d6d2fac2452f93e357. !Balance of receiving account Withdraw the token 1. After initiating a withdraw transaction, wait for the transaction data to be proven on L1. !Balance of receiving account !!! warning - As of 2024-03-27, there might be a small bug in the bridge UI which causes the transaction not to be claimable on L1 with the UI. - Essentially the bridge UI is selecting the wrong destination network so the proof will not validate. - That being said, it's possible to claim directly using the smart contracts. 2. Click Finalize. !Balance of receiving account Using cast to withdraw assets from the bridge The following work-in-progress cast script processes a bridge claim. Feel free to go through line-by-line and tweak where necessary. sh #!/bin/bash set -e Setup some vars for use later on The private key used to send transactions privatekey="0903a9a721167e2abaa0a33553cbeb209dc9300d28e4e4d6d2fac2452f93e357" The destination network (zero corresponds to L1/Ethereum) destinationnet="0" The address of the recipient destinationaddr="0x85dA99c8a7C2C95964c8EfD687E95E632Fc533D6" The bridge address bridgeaddr="$(kurtosis service exec cdk-v1 contracts-001 "cat /opt/zkevm/combined.json" | tail -n +2 | jq -r .polygonZkEVMBridgeAddress)" Grab the endpoints for l1 and the bridge service l1rpcurl=$(kurtosis port print cdk-v1 el-1-geth-lighthouse rpc) bridgeapiurl="$(kurtosis port print cdk-v1 zkevm-bridge-service-001 rpc)" The signature for claiming is long - just putting it into a var claimsig="claimAsset(bytes32[32],bytes32[32],uint256,bytes32,bytes32,uint32,address,uint32,address,uint256,bytes)" Get the list of deposits for the destination address echo "Getting the list of deposits..." curl -s "$bridgeapiurl/bridges/$destinationaddr?limit=100&offset=0" | jq > bridge-deposits.json cat bridge-deposits.json Filter the list of deposits down to the claimable txs that hasn't already been claimed and are destined for L1 echo "Filtering the list of deposits..." jq '[.deposits[] | select(.readyforclaim == true and .claimtxhash == "" and .destnet == '$destinationnet')]' bridge-deposits.json > claimable-txs.json cat claimable-txs.json Process all the claimable txs jq -c '.[]' claimable-txs.json | while IFS= read -r tx; do echo "Processing claimable tx..." echo "$tx" Use the bridge service to get the merkle proof of our deposit echo "Getting the merkle proof of our deposit..." currdepositcnt="$(echo "$tx" | jq -r '.depositcnt')" currnetworkid="$(echo "$tx" | jq -r '.networkid')" curl -s "$bridgeapiurl/merkle-proof?depositcnt=$currdepositcnt&netid=$currnetworkid" | jq '.' > proof.json cat proof.json Get our variables organized inmerkleproof="$(jq -r -c '.proof.merkleproof' proof.json | tr -d '"')" inrollupmerkleproof="$(jq -r -c '.proof.rollupmerkleproof' proof.json | tr -d '"')" inglobalindex="$(echo "$tx" | jq -r '.globalindex')" inmainexitroot="$(jq -r '.proof.mainexitroot' proof.json)" inrollupexitroot="$(jq -r '.proof.rollupexitroot' proof.json)" inorignet="$(echo "$tx" | jq -r '.orignet')" inorigaddr="$(echo "$tx" | jq -r '.origaddr')" indestnet="$(echo "$tx" | jq -r '.destnet')" indestaddr="$(echo "$tx" | jq -r '.destaddr')" inamount="$(echo "$tx" | jq -r '.amount')" inmetadata="$(echo "$tx" | jq -r '.metadata')" Generate the call data, this is useful just to examine what the call will look like echo "Generating the call data for the bridge claim tx..." cast calldata "$claimsig" "$inmerkleproof" "$inrollupmerkleproof" "$inglobalindex" "$inmainexitroot" "$inrollupexitroot" "$inorignet" "$inorigaddr" "$indestnet" "$indestaddr" "$inamount" "$inmetadata" Perform an ethcall to make sure the tx will work echo "Performing an eth call to make sure the bridge claim tx will work..." cast call --rpc-url "$l1rpcurl" "$bridgeaddr" "$claimsig" "$inmerkleproof" "$inrollupmerkleproof" "$inglobalindex" "$inmainexitroot" "$inrollupexitroot" "$inorignet" "$inorigaddr" "$indestnet" "$indestaddr" "$inamount" "$inmetadata" Publish the actual transaction! echo "Publishing the bridge claim tx..." cast send --rpc-url "$l1rpcurl" --private-key "$privatekey" "$bridgeaddr" "$claimsig" "$inmerkleproof" "$inrollupmerkleproof" "$inglobalindex" "$inmainexitroot" "$inrollupexitroot" "$inorignet" "$inorigaddr" "$indestnet" "$indestaddr" "$inamount" "$inmetadata" done Run the script with: sh sh scripts/bridge-manual-claim.sh You should see something like this: !Output from cast bridging script Confirm the claim went through using MetaMask and the bridge UI. !MetaMask confirmation !!! info Even though the bridge UI didn't allow me to claim, it does correctly show that a claim was executed. # fork.md: This document shows Polygon partners how to migrate an isolated CDK stack. Process to upgrade forks for isolated CDK chains In order to avoid reorgs and other undesirable scenarios, all L2 transactions must be verified before upgrading a fork. Verification means that all batches are closed, sequenced, and verified on L1. Follow the steps to verify all batches for upgrading. 1. Stop the sequencer. 2. Enforce the sequencer to stop at a specific batchnum. 1. In the statedb, get WIP batch number: SELECT batchnum, wip FROM state.batch WHERE wip IS true; Result = X (write down X for later) 2. Edit node config: 1. Sequencer.Finalizer.HaltOnBatchNumber = X+1 2. Sequencer.BatchMaxDeltaTimestamp = â€œ120sâ€ 1800s 3. SequenceSender.WaitPeriodSendSequence = "10s" 60s 4. SequenceSender.LastBatchVirtualizationTimeMaxWaitPeriod = â€œ30sâ€ 600s 3. Restart sequencer, sequence-sender. 4. Check sequencer halted when reaching batch X+1 (this is obvious in the logs). 5. Wait until all pending batches are virtualized (X): SELECT batchnum FROM state.virtualbatch ORDER BY batchnum DESC LIMIT 1; â†’ X 6. Wait until the aggregator has aggregated proofs for all the batches: 1. SELECT batchnum FROM state.verifiedbatch ORDER BY batchnum DESC LIMIT 1; â†’ Y (if Y == X) you can skip next steps until 3. Prepare (do not apply) new versions according to the version matrix 2. SELECT batchnum, batchnumfinal FROM state.proof WHERE NOT generating AND batchnum = Y AND batchnumfinal = X wait until this query returns a row, remove WHERE conditions to get a sense of progress. 7. Edit node config to force the aggregator into sending the already aggregated proof ASAP: Aggregator.VerifyProofInterval = "5mâ€. Then restart aggregator. 8. Wait until the proof is settled on-chain: SELECT batchnum FROM state.verifiedbatch ORDER BY batchnum DESC LIMIT 1; â†’ X 3. Prepare (do not apply) new versions according to the version matrix. 4. Stop all services (node, prover/executor, bridge). Update software 1. Start synchronizer's new version. 2. Wait until synchornizer receives a fork id event (check table state.forkid). 3. Edit node config file (node v0.6.2 version): 1. Sequencer.Finalizer.HaltOnBatchNumber = 0 4. Start all node components, executors, provers, and bridge with new versions. 5. Check batches â‰¥ X are virtualized and verified. 6. Edit new node config (restore previous values): 1. Aggregator.VerifyProofInterval = "25mâ€ restore previous value 2. Sequencer.BatchMaxDeltaTimestamp = â€œ1800sâ€ 3. SequenceSender.WaitPeriodSendSequence = "60sâ€ restore previous value 4. SequenceSender.LastBatchVirtualizationTimeMaxWaitPeriod = â€œ600sâ€ restore previous value 7. Restart sequencer, sequence-sender, and aggregator. # forkid-7-to-9.md: This document shows you how to migrate from fork 7 to fork 9 using the Kurtosis package. !!! tip These steps are similar to a production build, except you have to use a timelock contract to make the calls. Prerequisite steps and set up 1. Run a clean command to remove any lingering state: sh kurtosis clean --all 2. Downgrade all the necessary parameters to switch back to fork 7. Open the params.yml file and make the following changes: diff diff --git a/params.yml b/params.yml index 175619f..a72d452 100644 --- a/params.yml +++ b/params.yml @@ -29,13 +29,13 @@ args: deploymentsuffix: "-001" Docker images and repositories used to spin up services. - zkevmproverimage: hermeznetwork/zkevm-prover:v6.0.0 + zkevmproverimage: hermeznetwork/zkevm-prover:v4.0.19 - zkevmnodeimage: 0xpolygon/cdk-validium-node:0.6.4-cdk.2 + zkevmnodeimage: 0xpolygon/cdk-validium-node:0.5.13-cdk.3 - zkevmdaimage: 0xpolygon/cdk-data-availability:0.0.7 + zkevmdaimage: 0xpolygon/cdk-data-availability:0.0.6 zkevmcontractsimage: leovct/zkevm-contracts the tag is automatically replaced by the value of /zkevmrollupforkid/ @@ -160,7 +160,7 @@ args: zkevmrollupchainid: 10101 The fork id of the new rollup. It indicates the prover (zkROM/executor) version. - zkevmrollupforkid: 9 + zkevmrollupforkid: 7 3. Now kick-off a full redeploy: sh kurtosis run --enclave cdk-v1 --args-file params.yml --image-download always . 4. Confirm onchain that fork 7 is running: sh kurtosis files download cdk-v1 genesis /tmp/fork-7-test cast call \ --rpc-url "$(kurtosis port print cdk-v1 el-1-geth-lighthouse rpc)" \ "$(jq -r '.L1Config.polygonRollupManagerAddress' /tmp/fork-7-test/genesis.json)" \ "rollupIDToRollupData(uint32)(address,uint64,address,uint64,bytes32,uint64,uint64,uint64,uint64,uint64,uint8)" 1 Should you see 7 showing as the 4th parameter. 5. Send some test transactions to ensure batches are verified as expected. sh export ETHRPCURL="$(kurtosis port print cdk-v1 zkevm-node-rpc-001 http-rpc)" cast send --legacy --private-key "$(yq -r .args.zkevml2adminprivatekey params.yml)" --value 0.01ether 0x0000000000000000000000000000000000000000 cast rpc zkevmbatchNumber cast rpc zkevmvirtualBatchNumber cast rpc zkevmverifiedBatchNumber After a few minutes, the number of verified batches should increase (the first batch checked does not count). Make a clean stop of the sequencer 1. Before attempting the upgrade, we need to make a clean stop of the sequencer. To do this, pick a halting batch number by updating the node-config.toml file like this. Make sure to pick a batch number higher than the current batch number! sh cast to-dec $(cast rpc zkevmbatchNumber | sed 's/"//g') diff diff --git a/templates/trusted-node/node-config.toml b/templates/trusted-node/node-config.toml index 6c9b9fa..372d904 100644 --- a/templates/trusted-node/node-config.toml +++ b/templates/trusted-node/node-config.toml @@ -117,7 +117,7 @@ StateConsistencyCheckInterval = "5s" BatchMaxDeltaTimestamp = "20s" L2BlockMaxDeltaTimestamp = "4s" ResourceExhaustedMarginPct = 10 - HaltOnBatchNumber = 0 + HaltOnBatchNumber = 64 SequentialBatchSanityCheck = false SequentialProcessL2Block = true [Sequencer.StreamServer] 2. Re-run Kurtosis: sh kurtosis run --enclave cdk-v1 --args-file params.yml --image-download always . 3. Wait for the sequencer to halt and the verified batch to equal the latest batch and check the logs. sh kurtosis service logs cdk-v1 zkevm-node-sequencer-001 --follow You should see error logs that look like this: json {"level":"error","ts":1711481674.517157,"caller":"sequencer/finalizer.go:806","msg":"halting finalizer, error: finalizer reached stop sequencer on batch number: 64%!(EXTRA string=\n/home/runner/work/cdk-validium-node/cdk-validium-node/log/log.go:142 github.com/0xPolygonHermez/zkevm-node/log.appendStackTraceMaybeArgs()\n/home/runner/work/cdk-validium-node/cdk-validium-node/log/log.go:251 github.com/0xPolygonHermez/zkevm-node/log.Errorf()\n/home/runner/work/cdk-validium-node/cdk-validium-node/sequencer/finalizer.go:806 github.com/0xPolygonHermez/zkevm-node/sequencer.(finalizer).Halt()\n/home/runner/work/cdk-validium-node/cdk-validium-node/sequencer/batch.go:221 github.com/0xPolygonHermez/zkevm-node/sequencer.(finalizer).closeAndOpenNewWIPBatch()\n/home/runner/work/cdk-

validium-node/cdk-validium-node/sequencer/batch.go:163 github.com/0xPolygonHermez/zkevm-node/sequencer.(finalizer).finalizeWIPBatch()\n/home/runner/work/cdk-validium-node/cdk-validium-node/sequencer/finalizer.go:330 github.com/0xPolygonHermez/zkevm-node/sequencer.(finalizer).finalizeBatches()\n/home/runner/work/cdk-validium-node/cdk-validium-node/sequencer/finalizer.go:166 github.com/0xPolygonHermez/zkevm-node/sequencer.(finalizer).Start()\n)","pid":7,"version":"v0.1.0","stacktrace":"github.com/0xPolygonHermez/zkevm-node/sequencer.(finalizer).Halt\n\t/home/runner/work/cdk-validium-node/cdk-validium-node/sequencer/finalizer.go:806\ngithub.com/0xPolygonHermez/zkevm-node/sequencer.(finalizer).closeAndOpenNewWIPBatch\n\t/home/runner/work/cdk-validium-node/cdk-validium-node/sequencer/batch.go:221\ngithub.com/0xPolygonHermez/zkevm-node/sequencer.(finalizer).finalizeWIPBatch\n\t/home/runner/work/cdk-validium-node/cdk-validium-node/sequencer/batch.go:163\ngithub.com/0xPolygonHermez/zkevm-node/sequencer.(finalizer).finalizeBatches\n\t/home/runner/work/cdk-validium-node/cdk-validium-node/sequencer/finalizer.go:330\ngithub.com/0xPolygonHermez/zkevm-node/sequencer.(finalizer).Start\n\t/home/runner/work/cdk-validium-node/cdk-validium-node/sequencer/finalizer.go:166"} 4. Wait for the verified batch number to catch up to the trusted batch number: sh export ETHRPCURL="$(kurtosis port print cdk-v1 zkevm-node-rpc-001 http-rpc)" cast rpc zkevmbatchNumber cast rpc zkevmverifiedBatchNumber 5. When those two numbers are the same, stop the services that are going to be upgraded: sh kurtosis service stop cdk-v1 zkevm-executor-pless-001 kurtosis service stop cdk-v1 zkevm-node-aggregator-001 kurtosis service stop cdk-v1 zkevm-node-eth-tx-manager-001 kurtosis service stop cdk-v1 zkevm-node-l2-gas-pricer-001 kurtosis service stop cdk-v1 zkevm-node-rpc-001 kurtosis service stop cdk-v1 zkevm-node-rpc-pless-001 kurtosis service stop cdk-v1 zkevm-node-sequence-sender-001 kurtosis service stop cdk-v1 zkevm-node-sequencer-001 kurtosis service stop cdk-v1 zkevm-node-synchronizer-001 kurtosis service stop cdk-v1 zkevm-node-synchronizer-pless-001 kurtosis service stop cdk-v1 zkevm-prover-001 Smart contract calls 1. From another directory, make the required smart contract calls (this should not be done from the kurtosis-cdk directory): sh git clone git@github.com:0xPolygonHermez/zkevm-contracts.git pushd zkevm-contracts/ git reset --hard a38e68b5466d1997cea8466dbd4fc8dacd4e11d8 npm install printf "[profile.default]\nsrc = 'contracts'\nout = 'out'\nlibs = ['nodemodules']\n" > foundry.toml forge build 2. Deploy a new verifier. !!! tip This step isn't strictly necessary but good to do because in some cases you need a new verifier contract. sh forge create \ --json \ --rpc-url "$(kurtosis port print cdk-v1 el-1-geth-lighthouse rpc)" \ --private-key 0x12d7de8621a77640c9241b2595ba78ce443d05e94090365ab3bb5e19df82c625 \ contracts/mocks/VerifierRollupHelperMock.sol:VerifierRollupHelperMock > verifier-out.json 3. Create the PolygonValidiumStorageMigration contract: sh export ETHRPCURL="$(kurtosis port print cdk-v1 el-1-geth-lighthouse rpc)" ger="$(kurtosis service exec cdk-v1 contracts-001 "jq -r .polygonZkEVMGlobalExitRootAddress /opt/zkevm/combined.json" | tail -n +2)" pol="$(kurtosis service exec cdk-v1 contracts-001 "jq -r .polTokenAddress /opt/zkevm/combined.json" | tail -n +2)" bridge="$(kurtosis service exec cdk-v1 contracts-001 "jq -r .polygonZkEVMBridgeAddress /opt/zkevm/combined.json" | tail -n +2)" mngr="$(kurtosis service exec cdk-v1 contracts-001 "jq -r .polygonRollupManager /opt/zkevm/combined.json" | tail -n +2)" forge create \ --json \ --private-key 0x12d7de8621a77640c9241b2595ba78ce443d05e94090365ab3bb5e19df82c625 \ contracts/v2/consensus/validium/migration/PolygonValidiumStorageMigration.sol:PolygonValidiumStorageMigration \ --constructor-args $ger $pol $bridge $mngr > new-consensus-out.json 4. Add a new rollup type to the rollup manager: sh genesis="$(kurtosis service exec cdk-v1 contracts-001 "jq -r .genesis /opt/zkevm/combined.json" | tail -n +2)" cast send \ --json \ --private-key 0x12d7de8621a77640c9241b2595ba78ce443d05e94090365ab3bb5e19df82c625 \ $mngr \ 'addNewRollupType(address,address,uint64,uint8,bytes32,string)' \ "$(jq -r '.deployedTo' new-consensus-out.json)" \ "$(jq -r '.deployedTo' verifier-out.json)" \ 9 0 "$genesis" 'test!!!' > add-rollup-type-out.json 5. Get your new rollup type id: sh jq -r '.logs[0].topics[1]' add-rollup-type-out.json 6. Update the rollup with the id: sh rollup="$(kurtosis service exec cdk-v1 contracts-001 "jq -r .rollupAddress /opt/zkevm/combined.json" | tail -n +2)" cast send \ --json \ --private-key 0x12d7de8621a77640c9241b2595ba78ce443d05e94090365ab3bb5e19df82c625 \ $mngr \ 'updateRollup(address,uint32,bytes)' \ "$rollup" 2 0x > update-rollup-type-out.json 7. Verify the updated rollupid. Previously the 4th value was a 7 and now it should be a 9. sh cast call \ "$(jq -r '.L1Config.polygonRollupManagerAddress' /tmp/fork-7-test/genesis.json)" \ "rollupIDToRollupData(uint32)(address,uint64,address,uint64,bytes32,uint64,uint64,uint64,uint64,uint64,uint64,uint8)" 1 8. Set up the data availability protocol again: sh dac="$(kurtosis service exec cdk-v1 contracts-001 "jq -r .polygonDataCommittee /opt/zkevm/combined.json" | tail -n +2)" cast send \ --json \ --private-key "0x12d7de8621a77640c9241b2595ba78ce443d05e94090365ab3bb5e19df82c625" \ "$rollup" 'setDataAvailabilityProtocol(address)' $dac > set-dac-out.json Node upgrade At this stage, the smart contracts are upgraded. However, we still need to start the nodes again. !!! warning - This procedure is very sensitive. - Ensure the synchronizer starts first. We're going to revert the parameters back to the versions of the node that worked with fork 9, and only redeploy the CDK central/trusted environment. 1. Update the params.yml file as follows: sh yq -Y --in-place 'withentries(if .key == "deploycdkcentralenvironment" then .value = true elif .value | type == "boolean" then .value = false else . end)' params.yml 2. Remove the HaltOnBatchNumber setting that we added earlier. 3. Run Kurtosis to bring up the main node components. sh kurtosis run --enclave cdk-v1 --args-file params.yml --image-download always . 4. The core services are now running and we should be able to send a transaction and see the batch numbers moving through their normal progression. sh export ETHRPCURL="$(kurtosis port print cdk-v1 zkevm-node-rpc-001 http-rpc)" cast send --legacy --private-key "$(yq -r .args.zkevml2adminprivatekey params.yml)" --value 0.01ether 0x0000000000000000000000000000000000000000 cast rpc zkevmbatchNumber cast rpc zkevmvirtualBatchNumber cast rpc zkevmverifiedBatchNumber Troubleshooting 1. If you clone the zkevm-contracts repo in the same folder as kurtosis-cdk, you may see this error when you try to deploy the stack: sh Error: An error occurred running command 'run' Caused by: An error occurred calling the run function for command 'run' Caused by: An error starting the Kurtosis code execution '.' Caused by: Error uploading package '.' prior to executing it Caused by: There was an error compressing module '.' before upload Caused by: An error occurred creating the archive from the files at '.' Caused by: The files you are trying to upload, which are now compressed, exceed or reach 100mb. Manipulation (i.e. uploads or downloads) of files larger than 100mb is currently disallowed in Kurtosis. 2. You may also see errors like these: json {"level":"warn","ts":1711502381.03938,"caller":"etherman/etherman.go:661","msg":"Event not registered: {Address:0x1Fe038B54aeBf558638CA51C91bC8cCa06609e91 Topics: [0xd331bd4c4cd1afecb94a225184bded161ff3213624ba4fb58c4f30c5a861144a] Data:[0 0 0 0 0 0 0 0 0 0 0 0 90 104 150 169 140 75 124 126 143 22 209 119 199 25 161 216 86 185 21 76] BlockNumber:108 TxHash:0x1bb5e714dd96434ded2d818458cc517cf7b30f5787dbb3aedb667e5e3e96808e TxIndex:0 BlockHash:0xdf5850cd5a8975859595649a05ce245f02953e84af627e9b22a1f8381077f057 Index:0 Removed:false}","pid":7,"version":"0.6.4+cdk"} You can check them directly from the rpc: sh cast logs \ --rpc-url "$(kurtosis port print cdk-v1 el-1-geth-lighthouse rpc)" \ --address 0x1Fe038B54aeBf558638CA51C91bC8cCa06609e91 \ --from-block 108 \ --to-block 108 You can reverse an event with the following script: sh cat compiled-contracts/.json | jq '.abi[] | select(.type == "event") | .type = "function"' | jq -s | polycli abi decode | grep d33 cast sig-event 'SetDataAvailabilityProtocol(address)' 3. In the above example, it looks like the unregistered event is a call to SetDataAvailabilityProtocol(address). # forkid-9-12.md: > ðŸ'¡ For CDK fork ID9 chains NOT attached to the AggLayer (Isolated), can ignore section 4. > ðŸ'¡ For CDK fork ID9 chains attached to the AggLayer, follow steps in sections 1 to 5. This is a coordinated effort between Polygon and the Implementation Provider. 1. Summary of the Procedure To initiate a CDK chain upgrade, the Implementation Provider can request support from Polygon by submitting the "Request Help for an Issue with an Existing CDK Chain" through the service desk.





CDK Service Desk **Example Request**

Polygon will then collaborate with the Implementation Provider to schedule the UTC timing and dates for the upgrade. This planning enables the Implementation Provider to schedule testnet and mainnet maintenance windows for the respective networks, ensuring proper communication and coordination with their communities. See Example Maintenance Communication to Network Partners Implementation Providers can prepare for the customer network chains. The high-level steps in the collaborative process are: 1. Implementation Provider halting the sequencer, 2. Polygon executes the upgrade transaction, 3. Implementation Provider upgrades components to Fork 12 stack, 4. Implementation Provider restarts the components. 2. CDK Components Versions Please read carefully to fully understand the new architecture before starting the process: https://docs.polygon.technology/cdk/getting-started/cdk-erigon/ The table below lists the CDK Fork ID 9 components and the new CDK FEP Fork ID 12 component. > ðŸ'¡ Please note: Latest CDK FEP Fork ID 12 components are listed in the Polygon Knowledge Layer | CDK Components | Fork ID 9 | CDK Components | Fork ID 12 | |---|--|---|---| | CDK Validium node
Sequence sender
Aggregator | 0.6.7+cdk.1 | CDK Erigon RPC & CDK node | cdk-erigon:v2.1.x | | | | CDK node
Sequence sender
Aggregator | cdk:v0.3.x | | Tx pool manager | zkevm-pool-manager | Tx pool manager | zkevm-pool-manager use latest tag | | Prover | v6.0.0 | Prover | zkevm-prover v8.0.0-RC14 | | CDK data availability | v0.0.7 | CDK data availability | cdk-data-availability use latest tag | | zkEVM rollup node | v6.0.0 | zkEVM rollup node | N/A | | Contracts | v6.0.0 | Contracts | zkevm-contracts | | Bridge service | v0.4.2-cdk.1 | Bridge service | zkevm-bridge-service | | Bridge UI | Polygon Portal | Bridge UI | Polygon Portal | 3. Implementation Provider Preparation Steps The Implementation Provider must prepare in advance for the upgrade to ensure a smooth transition from fork ID 9 to fork ID 12. Failure to complete these steps ahead of time could result in delays or even cancellation of the scheduled upgrade. 1. The Implementation Provider downloads CDK Fork 12 components in advance so they are ready to deploy. 2. Map to the latest prover files which can be found here: https://storage.googleapis.com/zkevm/zkproverc/v8.0.0-rc.9-fork.12.tgz 3. Scale up the number of provers in advance. It is recommended that you at least double the number of provers up and running for the scheduled upgrade maintenance window. - Ensure all (majority) of the network batches are verified before starting the upgrade process, otherwise there will be additional downtime as we wait for the network to be ready. 4. Run the CDK-Node aggregator component in sync-only mode in advance to populate its database. 5. Required Erigon pre-syncing: - Generate data stream files from the current cdk-validium-node database. - Tool and instructions can be found here: - https://github.com/0xPolygonHermez/zkevm-node/tree/develop/tools/datastreamer - Use the Erigon tool (from cdk-erigon repo) to serve the DS: - go run ./zk/debugtools/datastream-host --file /path/to/directory - Start CDK-Erigon, which reads from this datastream (provide zkevm.l2-datastreamer-url: 127.0.0.1:6900 in the config). - Wait for it to sync to the tip. - CDK-Erigon can be stopped. The generated files will be used later during the upgrade process. The whole process should look more or less like this: bash PREREQUISITES: Install GO 1.23 WORKDIR=/tmp cd $WORKDIR git clone https://github.com/0xPolygonHermez/zkevm-node.git cd $WORKDIR/zkevm-node/tools/datastreamer vim config/tool.config.toml Edit [StateDB] section with your StateDB credentials make generate-file cd $WORKDIR git clone https://github.com/0xPolygonHermez/cdk-erigon.git cd $WORKDIR/cdk-erigon go run ./zk/debugtools/datastream-host \ --file $WORKDIR/zkevm-node/tools/datastreamer/datastream Bring up Erigon pointing to that DS (localhost:6900 if running locally) and let it fully sync to the end of the DS. 4. Polygon Preparation Steps 1. Polygon will collaborate with the Implementation Provider to schedule the UTC timing and dates for the upgrade, incorporating required timelocks. 2. Polygon will set up Google Meet calls between Polygon and the Implementation Provider's engineers to conduct planned upgrades for both testnet and mainnet on agreed dates. 3. Polygon will prepare in advance and with agreed timelock: - Rolluptype for fork 12 - Upgrade transaction to fork 12 4. For chains attached to the Polygon Agglayer, Polygon will handle steps to upgrade the permissionless node. 5. See example communication that Implementation Providers can use to prepare their customer network partners and communities. 5. Operational Steps Please Note: To avoid creating reorgs or other unwanted situations, it's important that all L2 transactions are verified before performing a fork upgrade. This means all batches should be closed, sequenced, and verified on L1. Steps to Halt the Sequencer 1. Stop the sequencer. 2. Reconfigure the node to enforce sequencer stops at a specific batchnum: - Get the current batch number from StateDB: sql -- Note resulting value as X: SELECT batchnum, wip FROM state.batch WHERE wip IS true; - Edit node configuration: yaml SEQUENCER CONFIG Where X is the batch number obtained from the SQL query: Sequencer.Finalizer.HaltOnBatchNumber = X+1 Optional: Reduce batch time to avoid excessive downtime Sequencer.Finalizer.BatchMaxDeltaTimestamp = "120s" 1800s yaml SEQUENCE SENDER CONFIG Optional: Reduce sending time to avoid excessive downtime SequenceSender.WaitPeriodSendSequence = "10s" 60s SequenceSender.LastBatchVirtualizationTimeMaxWaitPeriod = "30s" 600s yaml AGGREGATOR CONFIG Recommended: Reduce verify interval to avoid excessive downtime Aggregator.VerifyProofInterval = "5m" - Restart the sequencer, sequence sender, and aggregator to apply these configs. - Check that the sequencer halts when reaching batch X+1. - Wait until all pending batches are virtualized and verified (X): sql -- Both queries should return X SELECT max(batchnum) FROM state.virtualbatch; SELECT max(batchnum) FROM state.verifiedbatch; 3. Stop all services (node, prover/executor, bridge). > ðŸ'¡ For an isolated chain not attached to the Agglayer, the chain admin can perform operational step 4 on their chainâ€™s rollup manager contract. Polygon is not involved. Please Note: Wait for Polygon to send the L1 transaction (tx) and confirm it. 4. Polygon: Upgrade the Smart Contract (multisig): - Upgrade rollup to fork 12. - Wait for the Tx to be finalized. Steps to Deploy CDK FEP Fork 12 Components 1. With the network stopped, repeat Erigon sync to get it fully synced to the current state. - This instance is ready to act as Sequencer and/or RPC. Clone the whole Erigon config/datadir as many times as instances are needed. Pick one to be the new Sequencer (by setting the environment variable CDKERIGONSEQUENCER=1), and configure all other instances (permissionless RPCs) to point to the Sequencer: yaml zkevm.l2-sequencer-rpc-url: "http://sequencer-fqdn-or-ip:8123" zkevm.l2-datastreamer-url: "sequencer-fqdn-or-ip:6900" 2. Start the stateless Executor. 3. Start the CDK-Erigon Sequencer. 4. Verify in the sequencerâ€™s logs that new blocks are being generated with fork ID 12. 5. Start the Pool Manager (if used/needed). 6. Start CDK-Erigon RPCs (if used/needed). 7. Start the Bridge. 8. Start the CDK aggregator and Sequence Sender components. 9. Start the stateless Prover. Polygon Steps for CDK Chains Attached to the Agglayer Polygon's DevOps team will be accountable for upgrading the Agglayer permissionless nodes during the upgrade process. Post-Upgrade Validations 1. Test batch lifecycle. 2. Test the bridge. Example Maintenance Communication to Network Partners There is a planned maintenance window upgrade of the xxxx network on the following dates. This is to upgrade the xxx network from Fork ID9 to Fork ID12. Maintenance Event: Â xxx Network Upgrade from Fork ID9 to Fork ID12 Date: TBD by Implementation Provider Time: 00:00 PM EDT / 00:00 PM UTC Duration: 2 Hours Things to Note: - This upgrade is from Fork ID9 to Fork ID12. The change log can be viewed here. - Partners should not update their nodes until after the xxx network upgrade is confirmed as complete. - New Node Version: cdk-erigon:v2.1.x Upgrade Instructions for Community Partners (Testnet/Mainnet) Update FROM node version 0.6.7+cdk.1 up to cdk-erigon:v2.1.x. Instructions to Update Nodes 1. Stop the RPC, Synchronizer, and Executor components. 2. Update the node (RPC and Synchronizer) to cdk-erigon:v2.1.x. 3. Update the Prover/Executor to v8.0.0-RC12-fork.12. 4. Start the components in this order: 1. Prover/Executor 2. Synchronizer 3. RPC # stack-components.md: Developers can use CDK to configure chains that run the Polygon zkEVM protocol in either rollup or validium mode. We refer to these chains, in either mode, as a CDK FEP. As part of its finality mechanism, a CDK rollup or validium configured with this mode utilizes the type of ZK-proofs referred to as full execution proofs. What is a full execution proof? A full execution proof (FEP) is a zero-knowledge proof attesting to the correctness of the chain's full state transition. That is, an FEP attests to the fact that the underlying VM (such as the Polygon zkEVM, Succinct's zkVM, or MoveVM) has executed all state transitions in accordance with specifications. CDK FEP components Next, we detail the architectural components of the CDK FEP mode. The table below lists the CDK FEP components and where you can find them. | Component | CDK FEP stack | Notes | |--|--|---| | Node
RPC and sequencer | cdk-erigon:v2.1.x | Customizable, commonly:
- Sequencer = 1 node
- RPC = multiple nodes | | Data availability | cdk-data-availability | Only for validium mode use latest tag | | Contracts | zkevm-contracts | Use Fork ID12 contracts | | CLI | cdk:v0.3.x | Included in CDK repo | | Sequence sender | cdk:v0.3.x | Included in CDK repo | | Aggregator | cdk:v0.3.x | Included in CDK repo | | Tx pool manager | zkevm-pool-manager | Use latest tag | | Prover | zkevm-prover | v8.0.0-RC14 | Component descriptions Here are brief descriptions of each CDK FEP component. - CDK Erigon node, a fork of erigon, that manages the following: - Multiple RPC nodes that provide common APIs for sending transactions. - Sequencer for executing transactions, and creating blocks and batches. - DAC: The Data Availability Committee, specifically for validium mode, is a set of trusted actors who keep custody of all transaction data, including monitoring and validating

hash values the sequencer sender proposes to publish on L1. - Contracts: Various smart contracts deployed on L1 for the full implementation and complete functionality of the Polygon zkEVM protocol: - PolygonRollupManager - PolygonZkEVMBridgeV2 - PolygonZkEVMGlobalExitRootV2 - FflonkVerifier - PolygonZkEVMDeployer - PolygonZkEVMTimelock - CLI tool: A single command line interface tool for abstracting away the complexity of deploying or configuring CDK components. - Sequence sender: For sequencing batches. - In the case of a rollup, the sequence sender sends batch data and the sequenceBatches transaction to L1. - In the case of a validium, the sequence sender sends batch data to the Data Availability Committee (DAC), requests for signatures from the DAC, and sends the sequenceBatchesValidium transaction to L1. - Aggregator: For facilitating proving and verification, fetching and providing batch data and witness to the prover. - Transaction pool manager: For storing transactions submitted by users. - Prover: A complex cryptographic tool capable of producing ZK-proofs of hundreds of batches, and aggregating these into a single ZK-proof which is published as the validity proof. Support services Bridge service: A backend service for enabling clients like the web UI to interact with the bridge smart contract by providing Merkle proofs. Bridge UI: The Polygon bridge portal which abstracts away the backend operations involved in bridge deposits and withdrawals. Recommended explorer service: [Blockscout](), which is an application that allows you to view, confirm, and inspect transactions on EVM chains, optimistic rollups and zk rollups. However, users may opt to use a different explorer service. # cdk-repo-reference.md: | Component | Description | |--|---| | CDK repo | CDK component including CLI, sequence sender, and aggregator. | | CD Erigon node | Erigon node implementation. | | CDK data availability | Data availability for validiums. | | CDK rollup contracts | Smart contracts repo. | | Data streamer | Data streamer repo. | | Transaction pool manager | Transaction pool manager repo. | | Prover | zkEVM prover implementation. | | Bridge service | Bridge service implementation for CDK zkEVM networks, moving to Polygon Portal. | # third-party-guides.md: --- hide: - toc --- This page links to full guides that some of our solution providers have developed, including quick starts and guides that explain how to spin up and deploy validium and rollup CDK stacks.

Gelato

Deploy a CDK with Gelato.

Gateway

Build a custom CDK chain with Gateway.

Zeeve

Deploy a custom CDK chain with Zeeve.

Check out the solution provider page to see more info on our partners. # validium-vs-rollup.md: Polygon CDK running in validium mode inherits the core functionalities of a zkEVM rollup node and adds a data availability layer. Key differences | | Rollup | Validium | | ------------------------ | -- | --- | | Node type | cdk-erigon:v1.2.24 | cdk-erigon:v2.1 Data availability | On-chain via L1 | Off-chain via a local option, or a DAC + DA node | | Components | zkEVM components\ | zkEVM components\ + PostgreSQL database + on-chain committees | | Contracts | zkEVM smart contracts

- PolygonZkEVM (main rollup contract)
- PolygonZkEVMBridge
- PolygonZkEVMGlobalExitRoot

| Validium-specific DAC contract

- CDKDataCommittee.sol
- CDKValidium.sol

| | Infrastructure | Standard infrastructure | Dedicated infrastructure for data availability layer and DACs | | Tx flow | All transaction data is published on L1 | Validium only publishes the hash of the transaction data to L1. The sequencer sends both the hash and the transaction data to the DAC for verification. Once approved, the hash+signatures are sent to the Consensus L1 contract of the validium protocol. | Security | High security due to on-chain data availability and zero-knowledge proofs. |Off-chain data availability can affect security if the sequencer goes offline or if DAC members collude to withhold state data. | | Gas fees | High, because all transaction data is stored on Ethereum. | Low, because only the hash of the transaction data is stored on Ethereum. | | Proof generation | Uses Prover to generate proofs of batched transactions for validation. | Uses Prover to generate proofs of batched transactions for validation. | | Final settlement | Transaction batches and their corresponding proofs are added to the Ethereum state. | The hash of transaction data and its proof are added to the Ethereum state, referred to as the consolidated state. | JSON RPC, Pool DB, Sequencer, Etherman, Synchronizer, State DB, Aggregator, Prover # index.md: --- hide: - navigation - toc ---

Innovation & design

The Polygon Knowledge Layer consists of technical documentation that developers need for building with Polygon protocols, products, and services.

We also publish resources necessary for learning about and contributing to Polygon technologies.

This section gives you a peek into the future: a community-driven vision centered on cutting-edge Web3 development.

AggLayer documentation

AggLayer solves blockchain fragmentation by enabling sovereign chains to securely share liquidity, users, and state.

The future of Polygon network

Upcoming updates and iterations across Polygon network

Polygon Type 1 Prover

Driven by innovation, the Polygon type 1 prover is fully Ethereum equivalent.

Polygon security

Polygon security, bug bounties, and more.

Plonky 2 & 3

State-of-the-art cryptography libraries for developers.

Understanding chain reorgs in PoS

Find out more about reorgs in PoS; why they happen and how they work.

plonky.md: Plonky 2 & 3 are active research efforts into the development of libraries containing performant cryptographic functions for use in zero-knowledge proof systems. Plonky was Polygon's original implementation of a zk-SNARK computational cryptographic library based on Plonk with some customizations. The original library was decommissioned in 2021 to be replaced by Plonky2. Plonky 2 Plonky2 is a performant Rust library of cryptographic functions that includes a SNARK implementation based on techniques from Plonk and FRI as well as tools such as Starky. The library has an emphasis on fast recursive techniques. Polygon's type 1 prover system uses the Plonky2 library. Examples The Polygon Plonky2 repo provides some example functions that you can try out: - factorial: Proving knowledge of 100 factorial. - fibonacci: Proving knowledge of the hundredth Fibonacci number. - rangecheck: Proving that a field element is in a given range. - squareroot: Proving knowledge of the square root of a given field element !!! tip "More info" - Check out the Plonky2 tutorial for more information on the library and how to use it. - Check out the Plonky2 whitepaper. Plonky 3 Plonky3 is an ongoing effort to provide a new and enhanced cryptographic library which aims to further improve on the speed and efficiency of recursive zero-knowledge proofs. It contains optimizations for newer CPU specifications. It implements polynomial IOPs, such as PLONK and STARKS, and commitment schemes such as Brakedown. Check out the Plonky3 README for an update on what is included. Head over to the Polygon Plonky3 repo and try it out yourself. !!! tip "Learn more about Plonky 1, 2, and 3" Watch the Polygon Zero team's introduction to Plonky on YouTube.
polygon-protocols.md: Polygon PoS Polygon PoS is a widely-adopted EVM-compatible sidechain designed for low transaction costs and fast transaction times. Where Polygon PoS is heading Here is how the proposed development, with consensus of the Polygon community, will be achieved: - Changes to network architecture. Becoming a zero-knowledge validium instead of a zk-rollup. - Changes to network token. - Introduction of a Staking Layer, which will eventually serve all L2 chains in the Polygon network. PIP-18 has suggested a phased approach for upgrading Polygon PoS to a zkEVM validium. This community discussion is guiding the technical specification of the potential upgrade. If this PIP is accepted, Polygon PoS will eventually plug into the Agglayer that will connect the entire Polygon network. !!! info "What's Agglayer?" AggLayer is the interoperability layer that EVM-compatible chains connect to, enabling features such as seamless and efficient cross-chain communication, unified liquidity, and more. Read more on the AggLayer in the Polygon blog. Phase 0 of PIP-18 includes: - The initiation of the POL upgrade - The upgrade from MATIC to POL as the native (gas) token for PoS - The adoption of POL as the staking token for Polygon PoS - The launch of the Staking Layer and migration of Polygon public chains to consequently leverage it. Polygon zkEVM Polygon zkEVM is a Type-3, EVM-equivalent ZK rollup, which maximizes network security by posting both ZK proofs and transaction data to Ethereum. Where Polygon zkEVM is heading Type 2: The primary goal for Polygon zkEVM is to become a full Type-2 ZK-EVM, supporting all nine precompiled smart contracts and all the opcodes. Security: The two near-term priorities for Polygon zkEVM are: - Adding support for a forced-transaction mechanism that will make the network censorship-resistant. - Lengthening the 10-day timelock for network upgrades. EVM-equivalence: As EIPs change and shape Ethereumâ€™s execution logic, Polygon zkEVM will evolve to maintain EVM-equivalence. The two most important forthcoming changes are: - EIP-4844: Expected in the Cancun hard fork, proto-Danksharding creates a low-cost alternative to CALLDATA, expected to greatly reduce transaction fees on rollups. - EIP-7212: A new precompiled contract will support signature verifications in the secp256r1 elliptic curve. Polygon zkEVM will continue to offer the strongest open source software security guarantees by making transaction data available (DA) on Ethereum. Eventually, Polygon zkEVM will plug into the Agglayer that will help connect the entire Polygon ecosystem. Polygon CDK Polygon CDK is a collection of open source software components for launching ZK-powered L2 chains on Ethereum. Currently, it supports two primary configurations, rollup mode and validium mode, with an additional customizable data availability committee (DAC) for validium chains. Where Polygon CDK is heading The next release of Polygon CDK, expected in Q1 2024, will support a wider set of features and configurations. These include: - The unified LxLy bridge: As the first phase of the Agglayer development, projects building with CDK will have the option to plug into the unified LxLy bridge that will, eventually, help connect the entire Polygon ecosystem. - Custom native (gas) tokens. - Custom allow-lists for smart contracts and transactions. - Support for modular, third-party data availability solutions. Polygon Miden Polygon Miden is a novel ZK rollup currently in development, designed as a zkVM. Powered by the Miden VM, Polygon Miden extends Ethereumâ€™s capabilities, offering enhanced throughput and privacy features beyond those of the EVM. Where Polygon Miden is heading The new version (v2) of Polygon Miden Alpha testnet was launched in Q2 2024, and the mainnet is expected to go live later in the year. Once Polygon Miden is on mainnet, it will plug into the Agglayer that will help connect the entire Polygon ecosystem. # welcome.md: The Polygon Knowledge Layer consists of two parts. Firstly, documents that developers need in order to build with Polygon protocols. Secondly, resources necessary for learning about Polygon technologies. This section offers a glimpse into the future, presenting a vision that is community-driven. The basics Polygon technologies will help developers build in an elastically scalable and unified ecosystem of ZK-powered Layer 2s on Ethereum, where users can create, program and exchange value. Polygon envisions a unified multichain ecosystem. A web of interoperable ZK-powered Ethereum L2s, with near-instant and atomic L2 <-> L2 transactions, designed to empower developers build without limitations. An ecosystem where developers can choose to build dApps, design and launch dedicated application-specific L2 chains, or migrate existing EVM Layer 1 chains to become L2s. The endgame is to allow developers to build in an environment that feels and functions more like the internet. This means a blockchain ecosystem that can scale without limit, seamlessly unified, and backed by the decentralization and security of Ethereum. Zero-knowledge is the key Fundamentally, building this web of ZK-powered L2s comes down to one challenge: trustless, off-chain computation. In order to scale Ethereum, one needs to preserve Ethereumâ€™s execution logic while making it more efficient. The best way to accomplish this goal is through zero-knowledge cryptography as it is capable of providing verifiable proofs that attest to the integrity of off-chain computations. Otherwise, scaling technologies often have to add additional socio-economic mechanisms to mediate off-chain computations, which often results in delayed settlement of transactions. Future iterations will focus on applying the open source, zero-knowledge scaling technology developed at Polygon Labs, and these advancements will enable Ethereum to scale to the limits of the internet.. # bugbounty.md: Immunefi https://immunefi.com/bounty/polygon https://immunefi.com/bounty/polygonzkevm/ Hexens https://r.xyz/program/f86864f2-8e7b-443a-84fe-cc0925f06221 HackenProof https://hackenproof.com/polygon-technology/polygon-pos HackerOne https://hackerone.com/polygon-technology # contact.md: Polygon Labs security organization is happy to talk to you. Feel free to reach out to us at: security@polygon.technology # disclosure.md: Have you discovered a security flaw or CVE related to our technology? Please notify us before informing the outside world, so that we can take action. This is known as â€˜responsible disclosureâ€™. Email us at security@polygon.technology, and use the PGP key below to encrypt any sensitive data. If you are looking for a reward or bounty related something you personally discovered, submit your request to one of our bug bounty programs. PGP public key -----BEGIN PGP PUBLIC KEY BLOCK----- Version: Keybase OpenPGP v2.0.76 Comment: https://keybase.io/crypto xsFNBGUT/qEBEADieeiOhgfHn8KtAz+r9zAig3aiBlj2CxBwATVYUK9jDm+4SdHu godygYTPfhToHlRv/yptDUNRIHbNeqg1iZNcsgi0+uSZ8fvmyfSbnZC3WMJSUE8z d+C9h6/Oh50YfHbPCUu3UtU4BWzhTrvhxdPcwq4epsguiolDEP4Um8B8iCbbfJGw zbguT8o4qg/EYyEI6fTtnAOHx6489GD/7b3WCgrVcy5s6XSXpw3X+eQE8B2nbWTg

Cahn6I5CPO/ZUhQ+Ma/3uRcLcQzhLC3IgTzjdTYHAbRgKurTPllZrOOAh4/FG4rY XzQRJ9DtxSvnTZXXlV5GE4Grfut9sHAVk/dN9CYMlgvohp5gutH+T9sUO2NEAlMu
S0slJctr0RCs+9Nl60DxtvBz3lMgJksiS/HugE34gEcVPr7QfklHXlqsnhbDJPNo U+DQdCyondqmWlXYhXYLMea70nKA6YyINcavoexb5DGn+Gy2cf+3co4prF/QblBr
eKy2MBGPvjXD/GC2QOS1YephQwICfFk+VoE9Gk3gIu6yXKEp+nteUGxgvCRuYF0Z HvixeNAvGchdxiCFoH6fMtrvutW0QOjGKLJu1p6aFkWR93Dd7gduJupwlt/RYDNG
7stxUZu9pVbL9ml8KPkGdOBlC8yPjCvWBeQwmCSMt1hPG1A1plPNSDKFywARAQAB zS5wb2x5Z29uLXNlY3VyaXR5IDxzZWN1cml0eUBwb2x5Z29uLnRlY2hub2xvZ3k+
wsF6BBMBCgAkBQJlE/6hAhsvAwsJBwMVCggCHgECF4ADFgIBAhkBBQkAAAAAAAoJ EIXYxnZoYpdPnSgP/0fcF7sksnjlgVHSRp+w3veWbPHDphpWZC4mhRDPHE77Elem
25kWB3k75KW5JhH/pCtimO5mfM8G2WKSH5rQ1FU1t58vZfH6iEJWYzEAObzWfvak QLXCJ3LV4AKAomIMNp7m/+ZtoD5qTE4HICITJFIVLcpu51AIhZuC757ivujuqLUK
PhzdDCC0a6JSlFunqi50ZBAEgdAuDgl0NHpYf09r3T5tw/0ByvBsjqxLYhe/Aazj yq5JVa6nV5ka8B/tIeP6kWEs1a82+RgyKHns+O4OFmZN7QxURXQHmaB/+tLoSNth
PBZs/W8GGICa3MDrlyDkCl+b+WleIGlDbTyOzk7FRq39OBo0VelIxML51I7wQjw8 07dt3bzO+c2vo87dXIOAXFbxHhL/nJyI83XwQ1hCwSRutb7P8j/VveFCfkIw6e/h
3lgW5lSgJUg3TSon+2RJaF+STCV1edTYM1/YYtqiIJidyC/McIUZTH5oMLNbECTy gF9dfk15gIIa6d+26zhw+q/jmx534JMdyv6a4n9WvE/diDCq0woEetoMfF95qP0j
F3qqQvw0yhMmMg8sx+7br40SEFbeEG5SoIVAuMQtxBDCQzaEuLtUjX3iJqv9aP1V vqyXZqJjjLZEm9BryBWsmYT/5UHMegcElHkcEYRIG8BSyMhklNzXTtUUbzsVzsFN
BGUT/qEBEADAB1oI85wHe84DNKU95lzuZ/DZx8sbTT1GE+N4scgxAd8knJLof/a+ lU7yxuVON0iEnq08r1LHI5cEsdAWu8v8Yv0hbXR3jzryZXIjkEi6mXhujzzw/Ghz
ACXvwS6RuYNZEemDg/cYN6N8i/olGEZNuDmKbD9nVoy0eEmefXD/JQGiLFJ7yvei ou7vwHP6LxzBkcb/g4n4i1TBLGqeJJGtQrtwVlwrD6NE3/ERO8U7biZ3TD6jLBWm
fWSI5IYN+4PUrCttLGri+TOwtVs1rl2bBMaj+InMm9bj5WjJF+XVKZQMoZln+Qn2 AkqT2PR9PaqrxCEuPehc/Iyfr1f4Fy3xHxkU5trh3e9TiAvGNQp8Pt1N0SAqZwu6
zg+0Nfh2ep/7SEeKpNEH7vj36DJoCXWaLuiD1GFW2QyyNslGoH/2+T0Oow03l5cA H45SEXyezzHxUKtjv5/EFbIFsPKbOGqqZqxc+X4JHbU+kCHzVL9XMak31uBZ1vD+
2oLWSbsnoCjHYyiJBhwsJ0NpIxfYQdTzQFtTk3vQACCF8BTMJ0+s+MAeFFeJC8by Ew6MXpHGLjONIbu659kteEojuJIGGdgYTKIF5/0zZdVtz8MqvSio1baZMhw9V9HI
nS0zcWWGH94/R0/lkoYqeEbUJ4bzMHI/YHB70hL1lV3CaVJ7eV2VYwARAQABwsOE BBgBCgAPBQJlE/6hBQkAAAAAAhsuAikJEIXYxnZoYpdPwV0gBBkBCgAGBQJlE/6h
AAoJEAZbSlMNXOyEN1YQAIs/t82MftZrLt6UKOlyKMG0mwXey/NTWYNs34J/+Eqj fY9f5PVHQmpm4keuYGvrqDAVXxqmFEgsbRR62QBSDIX1RtZnjZ3K0cfI6hDNgQiE
VWlDLaXIB+ewQBUKynjMP2blJCrT3/THcQVqhNRP+0I07Qt+UMOTfxJJL92sgsEW lm4b+5+0BZQDYApA7LzSeB3yOqWCBdBxYrVRFOAGE8WuorawaTJPVdXvuH1OrM7z
9SrfpANjX0/Azs4QoRzH3w0zKn03fJlLZ2CT8j+ZvIW3/rXFYQ8yt8pg2Fs6Fb5X PzFWuboAbnCmcTvGIW+05FeGcOqS5qQIPt4oMyanQ29/SLhnWule6KP3JjmRo1gZ
kCFgkCKpO36kw9Yd6Anuiau70Y+rOKQNRxD03DdDvQclAvTgOlej+fgFiC/nrM8q D05UdOaWD64ci2gb6OP+YyLeWfj2S2EKnBGCLwODIu10o9SyJ+DGIYn1cdjlAw3J
cVed3hBsMkE2D1+HAA6IpfqDr17hRxFYGEvmEaOI8LlCKuLbjkbPUHsVR0IKNn/t rE0sL6ZI2tBLIBX6C7fJBds4JwO+ClM5pOVaJrDKUvWSW7RZnejFOV3+TkHX3J5T
CiFzBUcoWncNAg6ikcRLPcNNbfV1HcY3AGqQgW29TowbgE+donrJJU6qvR0vsQp6 LiMQAMvEG6MqiJN6bLeWs2Sh87FZIgeacFYOVteAwXFRPgYYOYEPaKjtkAGxOTe2
AzvGnQY8Tu2jsJN2URPYwWrhEhX16jqOc+mCFRQBy4aNczMw2au8bhylG323mfxB ODzMKKXmBCFYSYSqIy86nrKgLoXWKmVOkfhRRboDoeBbsbH78P7LIYpODNZfbuu3
JlhJb5yTEbxdoqmc8af07YVzNi2wYemi0NWxhj2gOjwR7l24hKw9w++ILuiymCHw z/QE1myDAI1aLoF+IaA8vN929FkW7YQ7IEJhvdlCWY57bkwZKrBysth7Gwc8EEf5
Nt/9d2CoU2w9nRkg0w40ysOXDmHOoJ3vNK+6U0egZJ2qmqX77QSuKgVlKPjTdTjB 4dg2oqjfPyKFGi8yBgU7p4SHF8iTpQsUlMR1WtkVApUDsRhHmaxBvwxRmXL7Q+Zi
SmD9YW4/gG9MB4zvQzHXIKjyrWjQ8VtNqWy8mXpW4UbTC4ao/VRGtvKC6lE8z4Wm WD0ByLEAyTlxZZpl8EOppNStlnU/FvopSAc8tf+1GeAW5alqlx779k4zOmyAstw5
DqB86JFVJ6Q1VD4Z3DMUj1lXjmVl9FYbpaGuyiBrVVE6+b6+n5/Vd7MxzigpaHEl LYZgB/Wh+XOABv1JUwIVT+J+6fJXw6rnFfnoXYW40Yr9k9j0zsFNBGUT/qEBEADC
w4vhyYyLemG7AWiQYP7SD4v20gxZdf3J9bKko25s7iEerP1t3lKw2FjUNwH3ePcH 8MJnmOkHt3R+nNXIN90ZFkI72QxZBttQfCxP8hmuGrJ1T2rE/DBNI0msFaomS4yH
3CLMHn6SWyUxaz0mRkb5nhgfTh3L4Lr99HPR8wYBNkdqLzzlCRs9fzO5Ib4rasNl S5UF93C7O5LP17Q0AEwJJk1h8rn/2W9vwew5upR09Xu2kRZxTwhJgn9CQCtycyNu
kLUjAzdF9QPTPKr2KsLvM5V51oQBQjffcNKmUy/wrVcEIROaYjTW3nYDzvraB/rY t9uWgbFBKC2NaZdA6488WIMfnhDf6MAJ9fTL7X5bdo2Gu0BrlDK9+8UV2xEHhs8W
z0D6AiLjS9tDkQSIv8JA/lhu7KD6sCAXidHwa3sBr1mI1vut2VW3lHrrccBspvad +4S1AqTUnNo8NdOBGCk0UXkuhYkBko/wmYuF2AMxGnLGm3gogLMwGDORLcb53dYf
Y+Hff98KcIdMMrQI4njEPsS32Boj/Vmq1aSVry3UqbGaqAoeD2de6uNStE21YQ+L L4+WFfdFV6/DhggpxqcXFC9CumqqaABhGbgPb81JJXH0GQtPriE89sbOAzLpnEcD
OW3eaf69hhwn3wItJAwxNMnm/LAJTFLRwrWF5411fQARAQABwsOEBBgBCgAPBQJl E/6hBQkAAAAAAhsuAikJEIXYxnZoYpdPwV0gBBkBCgAGBQJlE/6hAAoJEGvxxkyI
0skB/jwP/iFfYseztT4d73hg5+zMTNDD4C2KQHwKcChL95p9ihc3VCrlRZyrEV8k hQ5GVb3pHaaNs2FCM71IWx7OQsW+I5Ki6Kz6cgwQXi4Uojq59gxs1pX1MDMzhK6C
C877i4Mm/3IatXTnqGBezIjXeJnF9TFoDRjDSqybPXaDkE4THyINmQ0jFEf6u/uB 7BzrZonLgC6JrrrqYeuwRV2XRNDLeu4hdlvy6HlzfsNT00R5hDhWaAcx9OkBmIiB
udkURWNtPTlFZqrg+H0kOrlpGKanJXh89CwXuXqb72HIuQlJq//KROIdo3z+a2Pb DX7/cZwY79AVUVsfj3HOk2BeCAsczR/djxhfXRc84zMf0hTQXHCpy6ICIghBVvx5
CA9iZZm6mtjlzv42Yl5NLcKiDf/ItKa5/GfjS9PVRsh55a8zxh6T8oWWZ0ZiIJw/ VMmA+DRYqsv9F83v0QiaiJhwCvmLM/f6ipuxocG5z0FuG2vrCnl+CNhi9bulqLuo
2Pl/H1VTnod7ez4cGu+a9943836HpESqhxBOLxmwhCd5chorlgKR2ihCbPZDuVPN baOIgul69hUUumZUVYFD22oKqZvbpjmGtvGdSrC8FrraZ0iXnhseT2efJdAUELIg
sZ9kgy1RG9VV3AQJ5cBTQswtlCzzyEJpyZi0m6jL3VcHullHEf8LAhwP/1y5bEDl 3EJapzt2/OyK4/kBUawHhiQlHaKXG6lXabB3dd2RGhWzicSm3UVWUmj8tfBnzX1Y
EvpshbWLgWNHrCIVRmNizav3ucBRAx9q1m38l2GsnhjpHwfw8r4JjQWI5QjaJap9 tXGa3QmxAaBdJwWTorA4ljHczLNx864r5g9dvKM0ZlN40/gsA8hwkSFUKVeE1Bpv
n3lGI4JhIS+UkiVCY898iWh91Y6ejfuwX7INs8+wbcl5siCJ/OZyFG3XMsBpHN2f d2LJEF+/NB9id+QUcmfQ5iVYrAEs0ijdsfJWhnQ04ey8Y4eBSc4R/60xBjMC1j9S
3aYRoKZ2BwFnBO03jWzJ+0diFI0IpptwxjpxZz94nHP28yzZo3yE7f39Yzxw4npm ShHGXLZY/KxmlRRK2vy5aEBTulVfPRfmvilhhOBB1D6L6GKTMrNKa6rhKu2G9LnS
ein+koIWj0cDrX70HVDTB3h7O8+He28585v3vk857wtRQlQsBBg7uxhVRsfeB6hr dRFO6MFchft3GAyQ+aqu9/7yz/wtiUyUDI8xzM20YasGWI4ZnphY1URU2l21mQpO
I+LkrR21rOViAoGmcaHFSjv5Jp7jJjy6OXtJpIPV9u6qC8iRYCLpwErGSZzDZiiS 67X82D4gEsDLNjvjV4l7cn+CenuCWV+WFAOE =dn2w -----END PGP PUBLIC KEY BLOCK----- # governance.md: Polygon Labsâ€™ security program is designed and implemented following the ISO/IEC 27001 standards, an internationally recognized framework for managing and securing sensitive information assets. By adhering to these standards, Polygon Labs demonstrates a strong commitment to the protection of data; ensuring that confidentiality, integrity, and availability are maintained at all times. The ISO 27001-based security program at Polygon Labs involves the establishment of an Information Security Management System (ISMS), which is a systematic approach to managing sensitive information and minimizing risk. This includes conducting regular risk assessments to identify, analyze, and evaluate potential threats and vulnerabilities, as well as implementing appropriate security controls and measures to mitigate those risks. In addition to risk assessments, Polygon Labsâ€™ ISMS incorporates a comprehensive set of policies, procedures, and guidelines that cover various aspects of information security, such as access control, incident management, and business continuity planning. Employee training and awareness programs are also an integral part of the security program, ensuring that staff members understand their roles and responsibilities in safeguarding the organizationâ€™s information assets. Polygon Labs has a security team led by a CISO reporting to founders. # hr.md: Polygon Labs supports onboarding and offboarding service providers by following a process that begins with each service provider receiving a preconfigured laptop that auto enrolls in one of our Mobile Device Management (MDM) systems. MDM supports control of application usage and enforces security policy requirements on approved operating system versions and patch requirements. User access to shared services and Polygon Labs-approved SaaS tools is secured by providing the least amount of privileges required for a service provider to perform their tasks. Privileges are role based and given to each service provider based on the functional team they are assigned to. Polygon Labs uses single sign-on technologies to automate the administration of user access and permissions across all its SaaS tools. Automating the provisioning and removal of usersâ€™ access privileges limits the risk of human error and supports efficient auditing procedures. When a service provider exits the company, HR changes their status in our HRIS system, automatically removing their access to our SSO integrated SaaS platforms, and IT is immediately notified to initiate the wipe and recovery of their corporate system. Security awareness training Polygon Labs uses a SaaS platform to provide an integrated approach to email and security awareness training for all of our service providers. All service providers are required to pass the training during their first weeks of service. The key features of the platform are: - Industry-specific modules: Reinforce critical concepts mapped to key industry standards and security frameworks, including ISO, NIST, PCI DSS, GDPR, and HIPAA. - Real-world assessment: Safely test service providers on real-world threats with de-weaponized phishing attacks. - Comprehensive reporting: Track primary indicators of risk across the awareness training platform and take remedial action with easily discernible user risk scores. - Integrated risk insight: Leverage real-world click behavior to identify high risk users. - Effortless administration: 12-month programs with rapid deployment. # infrastructure.md: Polygon network infrastructure security Polygon Labs has developed network infrastructure via smart contracts that automatically transfers assets to-and-from the Ethereum blockchain for both the Polygon PoS network and Polygon zkEVM scaling solution. This infrastructure implements a lock-and-mint architecture which results in assets being locked by the smart contract implementations. On behalf of the Polygon community and broader industry, Polygon Labs has implemented certain monitoring features over the network infrastructure to enhance security. Much of the security efforts noted here are rigorously applied to network infrastructure, including risk management, secure software development practices, auditing, vulnerability management, CI/CI, on-chain monitoring, and bug bounties. Monitoring The on-chain infrastructure is monitored for real-time events as a way to augment the application security efforts associated with software development (i.e. threat modeling, code auditing, library and supply-chain risk, and bug bounties). The real time monitoring includes both on-chain machine learning models to detect unknown threats in real-time, as well as empirical rule-based algorithms to capture known adversarial or error scenarios. The monitoring infrastructure was developed both in-house, and by vendors as needed, to augment our capabilities in specific analysis areas. Any adverse events detected by our models and tools are evaluated, triaged and, if necessary, escalated to the proper team for further analysis. The monitoring process is integrated with our enterprise incident response process. Multisig security Specific requirements are followed by any Polygon Labs employee that is a signer on a multisig contract, which are used for various security reasons. Multisigs consist of Safes (previously Gnosis Safes) and other smart contract multisig implementations. Hardware wallets are hardware-based cold storage, such as Trezor or Ledger devices that store private keys and enable signing multisig transactions offline. Signer multisig requirements include: - Hardware wallet: Polygon Labs requires cold storage from an accepted vendor dedicated for company official use only and secured by a PIN. - Hot wallets: Hot wallets are not allowed for use on Polygon Labsâ€™ multisigs. - Corporate workstation: Signing must be performed from a company system managed by our enterprise mobile device management (MDM) platform complete with anti-virus (AV) and endpoint detection and device (EDR). - Clean key: All signers are required to create a clean key that has never been exposed to a hot wallet. - Mnemonic storage: Polygon Labs mandates safe storage of mnemonic passphrases and provides guidance to its employees. - Secure communication: All multisig signing events are coordinated using Polygon Labsâ€™ accepted communication protocols for multisigs. All corporate multisigs are monitored 24/7 by the Polygon security team. # operations.md: Logging Polygon Labs uses a variety of SaaS and bespoke infrastructure. Where audit logs are provided by those services, they are collected into a centralized repository and stored for a certain period of time to support internal operations should a security incident arise. Logs are reviewed automatically for anomalies to feed Polygon Labsâ€™ threat detection models. Monitoring Polygon Labs relies on a variety of sources that generate alerts for potential security incidents. Those sources include, but are not limited to, Google Workspace, Falcon CrowdStrike, AWS GuardDuty, GCP Security Command Center, Cloudflare, and Okta. Every system with built-in anomaly or threat detection directs its findings to a centralized SIEM, Coralogix, for our security analysts to review. Polygon Labs has security analysts distributed globally to help ensure timely triage of security alerts. Incident response Polygon Labs has established an incident response process that is modeled on industry best practices. We designate key people to act as subject matter experts to join the incident response team as needed and depending on the nature of a given cyber security incident. We also use third-party agencies to complement our incident response team from top tier security vendors. The lifecycle of a cyber security incident begins with detection and discovery. At Polygon Labs, we use a variety of tools such as anti-virus, endpoint detection and response, network intrusion detection, phish screening, and anomaly detection to help ensure we identify potential cyber security events early. We also provide our service providers and community with mechanisms to proactively report suspicious activity; including a ticketing system, instant messaging channels, and a dedicated phone number for emergencies. When an incident is identified, the security operations team performs triage and draws on our roster of subject matter experts to help with investigation and analysis. If an incident is declared a true positive we move from analysis to containment, remediation, and recovery. Polygon Labs carefully considers when, how, and who to communicate with during incident response. Impacted stakeholders are sent notifications in a timely manner to ensure they can take reasonable steps to protect their information if necessary. In order to ensure the incident response process remains relevant, we conduct regular incident response exercises if no real security incident has occurred after a given period. Authentication & access control Polygon Labs establishes standards for authentication and access control in its information security policy and information security standards documents. To ensure the security of our corporate systems, all service providers must adhere to strict authentication and authorization requirements. These may include, but are not limited to, usage of complex passwords, which should be changed regularly according to industry standards and two-factor authentication, together with single sign on is mandatory for accessing sensitive systems. Default, shared, or easily guessable passwords are strictly prohibited. Polygon Labs performs entitlement reviews for sensitive systems on a regular basis. Where applicable and available, systems are accessed via single sign-on (SSO). # overview.md: At Polygon Labs, ensuring the security of our information systems, and safeguarding sensitive data, is of paramount importance. We prioritize security throughout every facet of our operations, from implementing robust security policies and guidelines to adopting industry best practices, such as ISO 27001 as a baseline for our security program and adhering to the OWASP recommendations for secure software development. We invest heavily in continuous security training for our employees, keeping them informed of emerging threats and equipping them with the necessary skills to protect our assets. In addition, we embrace a proactive approach by incorporating security-by-design principles and using state-of-the-art security tools to detect and mitigate vulnerabilities at every stage of the development lifecycle. Commitment to security Dedicated security team Security comes first at Polygon Labs and our commitment is proven by our in-house security team of 10+ full-time security engineers & leaders. The team remains involved in the web3 space and, together with other major organizations, is driving new innovations and best practices for all. Continuous monitoring On behalf of the community, Polygon Labs monitors certain blockchain infrastructure for suspicious activities. Polygon Labsâ€™ in-house security team works alongside Polygon Labs engineers and other industry experts to stay updated on known vulnerabilities in the space. Periodic security assessments Polygon Labs periodically assesses the security of the software it develops and other applications through extensive internal testing and external engagements, such as audits and penetration testing. All software and applications have been assessed multiple times to date. Security assessments evolve as the industry matures. Bug bounty program Developers working on Polygon protocols enjoy ongoing bug bounty programs on leading platforms with rewards of up to $1M for reported vulnerabilities in Polygon network infrastructure. It is among the most significant bounty programs in the web3 community. Enable developer community Polygon Labs is committed to enabling the developer community to surface vulnerabilities and patch them before they are exploited. Polygon Labs has a strict focus on security in the development lifecycle, heavily testing all code and following best practices and standards such as the Secure Software Development Lifecycle. Conclusion In summary, our organization is one of the leaders in the web3 security sector. Our distinction arises from our commitment to not only implementing the best security practices but also dedicating substantial resources to this endeavor. Through tireless efforts, we have cultivated a culture of continuous improvement. Our success is related to our dedication to rigorous performance measurements. This steadfast focus on self-assessment empowers us to not only maintain the highest standards but to push the boundaries of excellence in web3 security. As we stride forward, we are driven by the belief that leadership is not just a position, but a relentless pursuit. Our unceasing investments in both human and technological resources, combined with our unwavering dedication to improvement, have positioned us at the forefront of the web3 security domain. Our journey towards excellence continues through our unassailable commitment to safeguarding the web3 landscape. # reports.md: Polygon Labs periodically assesses the security of different technology and applications through extensive internal testing and external (public & private) engagements; such as code reviews, security audits, red team assessments and penetration testing. All technology and applications have been assessed multiple times to date. Security assessments continue as the network matures. The following information relates to the latest available (and public) external assessments and certifications: ISO/IEC 27001:2022 certification Polygon Labs was awarded ISO 27001 certification in March of 2024. Certificate https://www.schellman.com/certificate-directory (search for "Polygon Labs") Scope The scope of the ISO/IEC 27001:2022 certification is limited to the information security management system (ISMS) supporting Polygon Labs' business of developing blockchain scaling solutions; which includes personnel, policies, procedures, standards, systems, endpoint devices, applications, data, and controls in accordance with the statement of applicability, version 1.2, dated October 11, 2023. Portal - Penetration testing assessment by Cobalt.io in Jan 2023. POL Token - Security audits by ChainSecurity & SigmaPrime: https://github.com/0xPolygon/pol-token/tree/main/audit. POS - Bor/Heimdal milestones audit by Least Authority: https://github.com/maticnetwork/bor/blob/develop/audit/audit-feature-milestones.pdf. - POS portal audits: https://github.com/maticnetwork/pos-portal/tree/master/audits. - POS contracts: https://github.com/0xPolygon/pos-contracts/tree/main/audit. Unified bridge - Security audits by Sigma Prime, Hexens & Spearbit: https://github.com/0xPolygonHermez/zkevm-contracts/tree/main/audits zkEVM - zkEVM-Rom security audit by Verichains in Jan 2023: https://github.com/0xPolygonHermez/zkevm-rom/tree/main/audits. - Security audits by Hexens & Spearbit: https://github.com/0xPolygonHermez/zkevm-rom/tree/main/audits. CDK Most of components have been reviewed as part of zkEVM's audits. - Bridge service: Penetration testing assessment by Cobalt.io in March 2023. - Bridge UI: Penetration testing assessment by Cobalt.io in March 2023. Zero - Security audits by Least Authority: https://github.com/0xPolygonZero/plonky2/tree/main/audits. # risk.md: Polygon Labs' approach to security risk management consists of a process driven approach using a risk management framework to systematically assess, manage, and mitigate risk; while aligning security controls to international compliance requirements. The program provides a real-time view of Polygon Labs' current security posture while informing the security roadmap as new controls are continuously implemented and re-assessed to adjust for a dynamic threat environment. Some key initiatives and aspects of the Polygon Labs Infrastructure Information Risk

Management Program include: Risk assessment The objective of a risk assessment is to enumerate threats, identify vulnerabilities, determine the organizational impact of a threat along with the likelihood of the threat occurring. This process informs other aspects of the risk management process, including assessing the implementation or enhancement of security controls and measuring organizational residual risk. A risk assessment provides a risk-based approach to systematically identifying high-risk areas of focus. Standardized controls While every situation is unique, we understand the benefit of best practices. For the cloud, we use the CIS v8 control set. Residual risk Any risk identified at risk assessment requires analysis and a plan of action (i.e. Reduce, Avoid, Transfer, Accept). The implementation of mitigating controls is driven by a cost-benefit analysis of the impact and mitigation using both the Factor Analysis of Information Risk (FAIR) and qualitative approaches. FAIR is an internationally accepted standard which quantifies risk in financial terms. Compliance Polygon Labs maps security controls to various compliance initiatives such as ISO 27002. ISO 27002 controls provide near-universal mapping to other compliance requirements. Security roadmap The risk management program continuously maps to the controls implementation framework as we adjust to new threats and an evolving internal product suite. Monitoring We strive for continuous monitoring for situational awareness and security posture management. Benchmarks Where possible, we apply benchmarks that provide specific and measurable metrics for compliance with control requirements and policies. This provides KPIs that guide implementation efforts which feed into our residual risk and any continuous risk assessment activities. We strive to automate metrics; for example using scanning tools that directly measure control compliance to benchmarks. The risk management framework is supported by various internal and external resources including penetration testers and auditors for independent verification and validation. # sdlc.md: Polygon Labs engineering teams are trained and instructed to use secure coding guidelines and follow industry standards for secure development, such as OWASP, which provides guidelines, tools, and resources to help our developers identify and mitigate security risks. Starting with activities such as threat modeling and risk assessments, Polygon Labs can systematically identify and prioritize potential security threats and vulnerabilities in systems and applications. These proactive measures enable us to allocate resources effectively, focusing on areas that pose the greatest risks. Continuous integration and continuous deployment (CI/CD) activities are enforced in all code repositories, which implement automated security testing and scanning tools into the CI/CD pipeline to detect vulnerabilities early in the development process. Following development and testing phases, all applications expected to go into production are further tested via internal or external assessments such as penetration testing, security audits, and bug bounty programs. These efforts help validate the effectiveness of our security controls, detect weaknesses, and address them before they can be exploited by malicious actors. # smartcontracts.md: Polygon Labs takes the following approach in regards to smart contract security. Secure coding guidelines Smart contract codebases must be organized, readable, and understandable across multiple developers and project phases. Engineers tasked with developing such codebases follow industry-standard, secure coding practices and style guides, such as https://docs.soliditylang.org/en/latest/style-guide.html and https://github.com/coinbase/solidity-style-guide. Internal assessments Polygon Labs application security teams are composed of senior & staff security engineers that perform internal reviews on all code developed. This in-house expertise allows us to follow standard methodologies for assessments using available tooling for static analyzing, line-by-line manual reviews, fuzzing, and formal verification where applicable. External assessments After internal reviews, and based on a risk assessment, new smart contracts and major changes/upgrades are sent to reputable, tier 1 security consultancy organizations for a formal external security assessment. Polygon Labs periodically rotates vendors to ensure an unbiased view of the code. Polygon Labs' public reports are located here: Security reports. # vulnerability.md: Our vulnerability management lifecycle takes the output of secure development lifecycle activities, together with results from vulnerability tools such as security scanners, to ensure effective reduction of the risk they present. Vulnerabilities are sent to a centralized issue and findings tracker ensuring that all identified vulnerabilities are effectively managed. This system enables appropriate validation, triage, assessment, and remediation/mitigation of vulnerabilities by assigning them to relevant teams and stakeholders. Polygon Labs establishes a clear workflow and procedures to prioritize and address issues based on their severity, potential impact, and exploitability. Polygon Labs maintains open communication channels with vendors and security researchers, enabling us to stay informed of newly discovered vulnerabilities, patches, and updates. This collaboration significantly contributes to maintaining a secure environment by ensuring that systems and applications are up-to-date and protected against known threats. All these activities, and others, are part of our robust vulnerability management lifecycle, which effectively reduces the risks associated with security vulnerabilities and strengthens the overall security posture. # index.md: --- hide: - toc ---

Polygon Miden

Software in development

Polygon Miden is a zero-knowledge rollup for private, high-throughput applications.

Getting started

Follow the Miden getting started guide to get up and running.

Miden architecture

Learn more about the Miden architecture.

Roadmap

Check out the Miden roadmap for the latest updates.

index.md: --- hide: - toc ---

Polygon PoS

Polygon PoS is an EVM-compatible, proof-of-stake sidechain for Ethereum, with high throughput and low costs.

Get started with PoS

Get started with building on Polygon PoS.

Run a PoS node

Find out the differences between types of PoS nodes and how to set up, run, and deploy them.

PoS architecture

Find out about the PoS architecture and its Heimdall and Bor layers.

overview.md: The Polygon Proof-of-Stake (PoS) network is designed to address scalability challenges within the Ethereum ecosystem. It operates as an EVM-compatible Layer-2 (L2) scaling solution for Ethereum, enhancing its throughput while also significantly bringing down gas costs, i.e., transaction fees. Dual layer architecture Polygon PoS is a Proof-of-Stake Layer-2 (L2) network anchored to Ethereum, and is composed of the following two layers: - Heimdall layer, a consensus layer consisting of a set of proof-of-stake Heimdall nodes for monitoring staking contracts deployed on the Ethereum mainnet, and committing the Polygon PoS network checkpoints to the Ethereum mainnet. The new version of Heimdall is based on CometBFT. - Bor layer, an execution layer which is made up of a set of block-producing Bor nodes shuffled by Heimdall nodes. Bor is based on Go Ethereum (Geth). Transaction lifecycle The following cyclical workflow outlines the operational mechanics of today's Polygon PoS architecture: 1. User initiates transaction: On the Polygon PoS chain, typically via a smart contract function call. 2. Validation by public checkpoint nodes: These nodes validate the transaction against the Polygon chain's current state. 3. Checkpoint creation and submission: A checkpoint of the validated transactions is created and submitted to the core contracts on the Ethereum mainnet every 30 minutes or so. 4. Verification by core contracts: Core contracts verify checkpoint validity 5. Transaction execution: Upon successful verification, the transaction is executed and state changes are committed to Polygon PoS. 6. Asset transfer (optional): If needed, assets can be withdrawn to the Ethereum mainnet via the exit queue in the core contracts. 7. Cycle reiteration: The process can be initiated again by the user, returning to step 1. !!! info "Checkpoint verification and L2 transactions" Checkpoint verification plays an important role in ensuring the security of the PoS network, especially in the case of bridging, and other cross-chain transactions. In the case of simple transactions such as an L2 to L2 token transfer, the state finality is near instantaneous. Core contracts on Ethereum Ethereum serves as the foundational layer upon which Polygon's PoS architecture is built. Within the Ethereum ecosystem, a set of core contracts play an important role connecting Polygon PoS to Ethereum. These core contracts are responsible for a range of functionalities, from anchoring the Polygon chain to handling asset transfers. The core contracts on the Ethereum mainnet incorporate a key feature for security and functionality: the exit queue. The exit queue manages the safe and efficient transfer of assets back to the Ethereum mainnet, allowing users to seamlessly move assets between the Polygon PoS chain and Ethereum without compromising data integrity or security. Public checkpoint nodes Public checkpoint nodes serve as validators in the Polygon PoS architecture. They perform two primary functions: transaction validation and checkpoint submission. When a transaction is initiated on the Polygon PoS chain, these nodes validate the transaction against the current state of the Polygon chain. After validating a set number of transactions, these nodes create a Merkle root of the transaction hashes, known as a "checkpoint," and submit it to the core contracts on the Ethereum mainnet. The role of these nodes is crucial as they act as a bridge between the Ethereum mainnet and the Polygon PoS chain. They ensure data integrity and security by submitting cryptographic proofs to the core contracts on Ethereum. Upcoming developments Originally launched as Matic Network in June 2020, Polygon PoS has undergone numerous upgrades since its inception. Initially designed to scale Ethereum through a sidechain, a new proposal on the Polygon forum suggests upgrading Polygon PoS into a zero-knowledge (ZK)-based validium on Ethereum. Polygon PoS will soon adopt the execution environment of Polygon zkEVM along with a dedicated data availability layer. This new architecture would be inherently interoperable with a broader network of ZK-powered Ethereum L2s via the AggLayer. Polygon PoS will continue to be the foundational infrastructure for a wide array of decentralized applications and services. More details about the overarching vision of a unified ecosystem of L2s on Ethereum can be found in the innovation & design space. # overview.md: Architecture This section provides architectural details of Polygon PoS from a node perspective. Due to the proof-of-stake consensus, Polygon PoS consists of a consensus layer called Heimdall and execution layer called Bor. Nodes on Polygon are therefore designed with a two-layer implementation represented by Bor (the block producer layer) and Heimdall (the validator layer). In particular, and on the execution client side, it delineates on snapshots and state syncing, network configurations, and frequently used commands when running PoS nodes. On the consensus client side, one finds descriptions on how Heimdall handles; authentication of account addresses, management of validators' keys, management of gas limits, enhancement of transaction verifications, balance transfers, staking and general chain management. Architectural overview The Polygon Network is broadly divided into three layers: Ethereum layer â€" a set of contracts on the Ethereum mainnet. Heimdall layer â€" a set of proof-of-stake Heimdall nodes running in parallel to the Ethereum mainnet, monitoring the set of staking contracts deployed on the Ethereum mainnet, and committing the Polygon Network checkpoints to the Ethereum mainnet. Heimdall is based on Tendermint. Bor layer â€" a set of block-producing Bor nodes shuffled by Heimdall nodes. Bor is based on Go Ethereum. !Figure: Ethereum, Bor and Heimdall architecture Staking smart contracts on Ethereum To enable the Proof of Stake (PoS) mechanism on Polygon, the system employs a set of staking management contracts on the Ethereum mainnet. The staking contracts implement the following features: The ability for anyone to stake MATIC tokens on the staking contracts on the Ethereum mainnet and join the system as a validator. Earn staking rewards for validating state transitions on the Polygon Network. Save checkpoints on the Ethereum mainnet. The PoS mechanism also acts as a mitigation to the data unavailability problem for the Polygon sidechains. Heimdall: Validation layer Heimdall layer handles the aggregation of blocks produced by Bor into a Merkle tree and publishes the Merkle root periodically to the root chain. The periodic publishing of snapshots of Bor are called checkpoints. For every few blocks on Bor, a validator on the Heimdall layer: 1. Validates all the blocks since the last checkpoint. 2. Creates a Merkle tree of the block hashes. 3. Publishes the Merkle root hash to the Ethereum mainnet. Checkpoints are important for two reasons: 1. Providing finality on the root chain. 2. Providing proof of burn in withdrawal of assets. An overview of the process: A subset of active validators from the pool is selected to act as block producers for a span. These block producers are responsible for creating blocks and broadcasting the created blocks on the network. A checkpoint includes the Merkle root hash of all blocks created during any given interval. All nodes validate the Merkle root hash and attach their signature to it. A selected proposer from the validator set is responsible for collecting all signatures for a particular checkpoint and committing the checkpoint on the Ethereum mainnet. The responsibility of creating blocks and proposing checkpoints is variably dependent on a validatorâ€™s stake ratio in the overall pool. See also Heimdall architecture. Bor: Block production layer Bor is Polygon PoS's block producer â€" the entity responsible for aggregating transactions into blocks. Bor block producers are a subset of the validators and are shuffled periodically by the Heimdall validators. See also Bor architecture. # introduction.md: Bor is an integral component of the Polygon network that operates based on principles derived from the Clique consensus protocol, detailed in EIP-225. This consensus model is characterized by predefined block producers who collectively participate in a voting process to appoint new producers, taking turns in block generation. Proposers and producers selection Block producers for the Bor layer are a committee selected from the validator pool on the basis of their stake, which happens at regular intervals and is shuffled periodically. These intervals are decided by the validator's governance with regard to dynasty and network. The ratio of stake/staking power specifies the probability to be selected as a member of the block producer committee. Selection process 1. Validators are given slots proportionally according to their stake. 2. Using historical Ethereum block data as seed, we shuffle this array. 3. Now depending on Producer count(maintained by validator's governance), validators are taken from the top. 4. Using this validator set and Tendermint's proposer selection algorithm, we choose a producer for every sprint on Bor. Detailed mechanics of Bor consensus Validators in Polygon's Proof-of-Stake system In Polygon's Proof-of-Stake (PoS) framework, participants can stake MATIC tokens on a designated Ethereum smart contract, known as the "staking contract," to become validators. Active validators on Heimdall are eligible for selection as block producers through the Bor module. Span: Defining validator sets and voting power A span is a defined set of blocks, during which a specific subset of validators is selected from the broader validator pool. Heimdall provides intricate details of each span through its span-details APIs. Within a span, each validator is assigned a certain voting power. The probability of a validator being chosen as a block producer is directly proportional to their voting power. The selection algorithm for block producers is borrowed from Tendermint's consensus protocol. Sprint: Single block producer selection within a span Within a span, a sprint is a smaller subset of blocks. For each sprint, only one block producer is selected to generate blocks. The size of a sprint is a fraction of the overall span size. Bor also designates backup producers, ready to step in if the primary producer is unable to fulfill its role. Block authorization by producers Block producers in Bor are also referred to as signers. To authorize a block, a producer signs the block's hash, encompassing all components of the block header except the signature itself. This signature is generated using the secp256k1 elliptic curve algorithm and is appended to the extraData field of the block header. Each block is assigned a difficulty level. Blocks signed in-turn (by the designated producer) are given a higher difficulty (DIFFINTURN) compared to out-of-turn signatures (DIFFNOTURN). Handling out-of-turn signing Bor selects multiple backup producers to address situations where the designated producer fails to generate a block. This failure could be due to various reasons, including technical issues, intentional withholding, or other disruptions. The backup mechanism is activated based on a sequential

order of validators and a predefined delay known as the "wiggle" time. Wiggle time: Delay before backup production Wiggle time is the predefined delay a backup producer waits before starting to generate a block. This delay is calculated based on the last block's production time and a variable parameter known as Period. The wiggle time is dynamically adjusted based on the position of the backup producer in the validator sequence relative to the designated producer's position. Resolving forks with difficulty metrics The potential for forks arises when backup producers generate blocks due to delays in block propagation. Bor addresses this by selecting the fork with the highest cumulative difficulty, reflecting the sequence of in-turn block production. The difficulty of a block is determined based on the validator's turn to produce the block, with in-turn production assigned the highest difficulty. View change and span commitment At the end of each span, Bor undergoes a view change, fetching new producers for the subsequent span. This involves an HTTP call to the Heimdall node to retrieve new span data and a commitSpan call to the BorValidatorSet genesis contract. Block headers in Bor are also structured to include producer bytes, aiding in the fast-syncing process. State synchronization with the Ethereum chain Bor features a mechanism to relay specific events from the Ethereum chain to Bor. This involves: 1. Contracts on Ethereum triggering the StateSynced event via StateSender.sol. 2. Heimdall monitoring these events and proposing state changes through StateReceiver.sol. 3. Bor committing these state changes at the start of every sprint, ensuring synchronization with the Ethereum chain's state. This state sync process is a crucial aspect of maintaining consistency between the Ethereum and Bor chains, ensuring that relevant state changes on Ethereum are reflected in Bor's state. # network-config.md: This guide provides a detailed list of default ports used across Polygon nodes, including Bor and Heimdall. Understanding these ports is crucial for network configuration and effective communication between nodes. Bor node | Name | Port | Tags | Description | |-------------------------|-------|--------------------------|---| | Network Listening Port | 30303 | Public | Port used by Bor for peer connections and synchronization. | | RPC Server | 8545 | Can be Public, Internal | RPC port for sending transactions and fetching data. Heimdall uses this port to obtain Bor headers. | | WebSocket Server | 8546 | Can be Public, Internal | WebSocket port for real-time updates. | | GraphQL Server | 8547 | Internal | GraphQL port for querying data. | | Prometheus Server | 9091 | Can be Public, Monitoring | Prometheus APIs for Grafana data source. Can be mapped to ports 80/443 via an Nginx reverse proxy. | | Grafana Server | 3001 | Can be Public, Monitoring | Grafana web server. Can be mapped to ports 80/443 via an Nginx reverse proxy. | | Pprof Server | 7071 | Internal, Monitoring | Pprof server for collecting Bor metrics. | | UDP Discovery | 30301 | Can be Public, Internal | Default port for Bootnode peer discovery. | Heimdall node | Name | Port | Tags | Description | |-------------------------|-------|-------------------------|---| | Network Listening Port | 30303 | Public | Port used by Heimdall for peer connections and synchronization. | | RPC Ser 8545 | Can be Public, Internal | RPC port for sending transactions and fetching data. Heimdall uses this port to obtain Bor headers. | | WebSocket Server | 8546 | Can be Public, Internal | WebSocket port for real-time updates. | | GraphQL Server | 8547 | Internal | GraphQL port for querying data. | | Prometheus Server | 9091 | Can be Public, Monitoring | Prometheus APIs for Grafana data source. Can be mapped to ports 80/443 via an Nginx reverse proxy. | | Grafana Server | 3001 | Can be Public, Monitoring | Grafana web server. Can be mapped to ports 80/443 via an Nginx reverse proxy. | | Pprof Server | 7071 | Internal, Monitoring | Pprof server for collecting Heimdall metrics. | | UDP Discovery | 30301 | Can be Public, Internal | Default port for Bootnode peer discovery. | # state-sync.md: State sync is the native mechanism for a user in the Polygon PoS chain to read the latest Ethereum data. Validators on the Heimdall layer pickup the StateSynced event and pass it on to the Bor layer. The receiver contract inherits IStateReceiver, and custom logic sits inside onStateReceive function. This is the flow required from dapps / users to work with state-sync: 1. Call the smart contract function present here: https://github.com/maticnetwork/contracts/blob/19163ddecf91db17333859ae72dd73c91bee6191/contracts/root/stateSyncer/StateSender.sol#L33 2. Which emits an event called StateSynced(uint256 indexed id, address indexed contractAddress, bytes data); 3. All the validators on the Heimdall chain receive this event and one of them, whoever wishes to get the transaction fees for state sync sends this transaction to Heimdall. 4. Once state-sync transaction on Heimdall has been included in a block, it is added to pending state-sync list. 5. After every sprint on bor, the Bor node fetches the pending state-sync events from Heimdall via an API call. 6. The receiver contract inherits IStateReceiver interface, and custom logic of decoding the data bytes and performing any action sits inside onStateReceive function: https://github.com/maticnetwork/genesis-contracts/blob/master/contracts/IStateReceiver.sol How does state sync work? State management sends the state from the Ethereum chain to the Bor chain. It is called state-sync. State transfer from Ethereum to Bor happens through system call. Suppose, a user deposits USDC to the deposit manager on Ethereum. Validators listen to those events, validate, and store them in Heimdall state. Bor gets the latest state-sync records and updates the Bor state (mints equal amount of USDC on Bor) using a system call. State sender Source: https://github.com/maticnetwork/contracts/blob/develop/contracts/root/stateSyncer/StateSender.sol To sync state, the state sender contract calls the syncState method on Ethereum chain. jsx contract StateSender { / Emits stateSynced events to start sync process on Ethereum chain @param receiver Target contract on Bor chain @param data Data to send / function syncState (address receiver, bytes calldata data) external; } The receiver contract must be present on the child chain, which receives state data once the process is complete. syncState emits StateSynced event on Ethereum, which is the following: jsx / Emits stateSynced events to start sync process on Ethereum chain @param id State id @param contractAddress Target contract address on Bor @param data Data to send to Bor chain for Target contract address / event StateSynced (uint256 indexed id, address indexed contractAddress, bytes data); Once the StateSynced event is emitted on the stateSender contract on Ethereum, Heimdall listens to those events and adds to the Heimdall state after $2/3+$ validators agree on the. After every sprint (currently 16 blocks on Bor), Bor fetches a new state-sync record and updates the state using a system call. Here is the code for the same: https://github.com/maticnetwork/bor/blob/6f0f08daecaebbff44cf18bee558fc3796d41832/consensus/bor/genesiscontractsclient.go#L51 During commitState, Bor executes onStateReceive, with stateId and data as args, on target contract. State receiver interface on Bor receiver contract on Bor chain must implement following interface. jsx // IStateReceiver represents interface to receive state interface IStateReceiver { function onStateReceive(uint256 stateId, bytes calldata data) external; } Only 0x0000000000000000000000000000000000001001 â€" StateReceiver.sol, must be allowed to call onStateReceive function on target contract. System call Only system address,Â 2^160-2, allows making a system call. Bor calls it internally with the system address asÂ msg.sender. It changes the contract state and updates the state root for a particular block. Inspired byÂ https://github.com/ethereum/EIPs/blob/master/EIPS/eip-210.md.Â System call is helpful to change state to contract without making any transaction. State-sync logs and Bor block receipts Events emitted by system calls are handled in a different way than normal logs. Here is the code: https://github.com/maticnetwork/bor/pull/90. Bor produces a new transaction/receipt just for the client which includes all the logs for state-sync. Transaction hash is derived from block number and block hash (last block at that sprint): jsx keccak256("matic-bor-receipt-" + block number + block hash) This doesn't change any consensus logic, only client changes. ethgetBlockByNumber, ethgetTransactionReceipt, and ethgetLogs include state sync logs with derived. Note that the bloom filter on the block doesn't include inclusion for state sync logs. It also doesn't include derived transaction in transactionRoot or receiptRoot. # authentication.md: Heimdall's auth module is responsible for specifying the base transaction and account types for an application. It contains the ante handler, where all basic transaction validity checks (signatures, nonces, auxiliary fields) are performed, and exposes the account keeper, which allows other modules to read, write, and modify accounts. Gas and fees Fees serve two purposes for an operator of the network. 1. Fees limit the growth of the state stored by every full node and allow for general purpose censorship of transactions of little economic value. Fees are best suited as an anti-spam mechanism where validators are disinterested in the use of the network and identities of users. 2. Since Heimdall doesn't support custom contract or code for any transaction, it uses fixed cost transactions. For fixed cost transactions, the validator can top up their accounts on the Ethereum chain and get tokens on Heimdall using the Topup module. Types Besides accounts (specified in State), the types exposed by the auth module are StdSignature, the combination of an optional public key and a cryptographic signature as a byte array, StdTx, a struct that implements the sdk.Tx interface using StdSignature, and StdSignDoc, a replay-prevention structure for StdTx which transaction senders must sign over. StdSignature A StdSignature is the types of a byte array. go // StdSignature represents a sig type StdSignature []byte StdTx AÂ StdTxÂ is a struct that implements theÂ sdk.TxÂ interface, and is likely to be generic enough to serve the purposes of many types of transactions. go type StdTx struct { Msg sdk.Msg json:"msg" yaml:"msg" Signature StdSignature json:"signature" yaml:"signature" Memo string json:"memo" yaml:"memo" } StdSignDoc AÂ StdSignDocÂ is a replay-prevention structure to be signed over, which ensures that any submitted transaction (which is simply a signature over a particular byte string) will only be executable once on a Heimdall. go // StdSignDoc is replay-prevention structure. // It includes the result of msg.GetSignBytes(), // as well as the ChainID (prevent cross chain replay) // and the Sequence numbers for each signature (prevent // inchain replay and enforce tx ordering per account). type StdSignDoc struct { ChainID string json:"chainid" yaml:"chainid" AccountNumber uint64 json:"accountnumber" yaml:"accountnumber" Sequence uint64 json:"sequence" yaml:"sequence" Msg json.RawMessage json:"msg" yaml:"msg" Memo string json:"memo" yaml:"memo" } Account It manages addresses, coins and nonce for transactions. It also signs and validates transactions. Source: https://github.com/maticnetwork/heimdall/blob/master/auth/types/account.go#L32-L54 go type BaseAccount struct { Address types.HeimdallAddress json:"address" yaml:"address" Coins types.Coins json:"coins" yaml:"coins" PubKey crypto.PubKey json:"publickey" yaml:"publickey" AccountNumber uint64 json:"accountnumber" yaml:"accountnumber" Sequence uint64 json:"sequence" yaml:"sequence" } Parameters The auth module contains the following parameters: | Key | Type | Default value | | --------------------- | ------ | ----------------- | | MaxMemoCharacters | uint64 | 256 | | TxSigLimit | uint64 | 7 | | TxSizeCostPerByte | uint64 | 10 | | SigVerifyCostED25519 | uint64 | 590 | | SigVerifyCostSecp256k1 | uint64 | 1000 | | DefaultMaxTxGas | uint64 | 1000000 | | DefaultTxFees | string | "1000000000000000" | CLI commands Show account Use the following command to print account related data into Heimdall: bash heimdalld show-account Expected result: json { "address": "0x68243159a498cf20d945cf3E4250918278BA538E", "pubkey": "0x040a9f6879c7cdab7ecc67e157cda15e8b2ddbde107a04bc22d02f50032e393f6360a05e85c7c1ecd201ad30dfb886af12dd02b47e4463f6f0f6f94159dc9f10b8" } Account and coin details To display account details, coins, sequence and account number: bash heimdallcli query auth account 0x68243159a498cf20d945cf3E4250918278BA538E --trust-node Expected result: json address: 0x68243159a498cf20d945cf3e4250918278ba538e coins: - denom: pol amount: i: "1000000000000000000000" pubkey: "" accountnumber: 0 sequence: 0 Parameters Use the following command to print all params: go heimdallcli query auth params Expected result: go maxmemocharacters: 256 txsiglimit: 7 txsizecostperbyte: 10 sigverifycosted25519: 590 sigverifycostsecp256k1: 1000 maxtxgas: 1000000 txfees: "1000000000000000" REST APIs | Name | Endpoint | Description | | ----------------------- | ---------------------------------- | -- | | Account details | /auth/accounts/{address} | Returns all details for an address | | Account sequer details | /auth/accounts/{address}/sequence | Returns only necessary details for signing | | Auth params | /auth/params | Returns all params auth module uses | # balance-transfers.md: Heimdall's bank module handles balance transfers between accounts. This module corresponds to the bank module from the Cosmos SDK. Messages MsgSend MsgSend handles transfer between accounts in Heimdall. Here is a structure for transaction message: go // MsgSend - high-level transaction of the coin module type MsgSend struct { FromAddress types.HeimdallAddress json:"fromaddress" ToAddress types.HeimdallAddress json:"toaddress" Amount types.Coins json:"amount" } MsgMultiSend MsgMultiSend handles multi transfer between account for Heimdall. go // MsgMultiSend - high-level transaction of the coin module type MsgMultiSend struct { Inputs []Input json:"inputs" Outputs []Output json:"outputs" } Parameters The bank module contains the following parameters: | Key | Type | Default value | | ------------- | ---- | ------------- | | sendenabled | bool | true | CLI Commands Send balance The following command sends 1000 POL tokens to the specified address: bash heimdallcli tx bank send

1000pol --chain-id # chain-management.md: This document specifies an overview of the chain manager module of Heimdall. The chain manager module provides all necessary dependencies like contract-addresses, borchainid, and txconfirmationtime. Other parameters can be added to this later on. Params are updated through the gov module. Types The chain manager structure on Heimdall looks like the following: go type ChainParams struct { // BorChainID is valid bor chainId BorChainID string json:"borchainid" yaml:"borchainid" // PolTokenAddress is valid POL token address PolTokenAddress hmTypes.HeimdallAddress json:"poltokenaddress" yaml:"poltokenaddress" // StakingManagerAddress is valid contract address StakingManagerAddress hmTypes.HeimdallAddress json:"stakingmanageraddress" yaml:"stakingmanageraddress" // RootChainAddress is valid contract address RootChainAddress hmTypes.HeimdallAddress json:"rootchainaddress" yaml:"rootchainaddress" // StakingInfoAddress is valid contract address StakingInfoAddress hmTypes.HeimdallAddress json:"stakinginfoaddress" yaml:"stakinginfoaddress" // StateSendedAddress is valid contract address StateSenderAddress hmTypes.HeimdallAddress json:"statesenderaddress" yaml:"statesenderaddress" // Bor Chain Contracts // StateReceiveAddress is valid contract address StateReceiverAddress hmTypes.HeimdallAddress json:"statereceiveaddress" yaml:"statereceiveaddress" // ValidatorSetAddress is valid contract address ValidatorSetAddress hmTypes.HeimdallAddress json:"validatorsetaddress" yaml:"validatorsetaddress" } CLI commands Parameters Use the following command to print all params: go heimdallcli query chainmanager params --trust-node Expected result: yaml txconfirmationtime: 12s chainparams: borchainid: "15001" poltokenaddress: "0x0000000000000000000000000000000000000000" stakingmanageraddress: "0x0000000000000000000000000000000000000000" rootchainaddress: "0x0000000000000000000000000000000000000000" stakinginfoaddress: "0x0000000000000000000000000000000000000000" statesenderaddress: "0x0000000000000000000000000000000000000000" statereceiveraddress: "0x0000000000000000000000000000000000000000" validatorsetaddress: "0x0000000000000000000000000000000000000000" REST APIs |Name |Method|URL | |----------------------|------|------------------| |Params |GET |chainmanager/params| All query APIs will provide response in the following format: json { "height": "1", "result": { ... } } # checkpoints.md: Checkpoints are vital components of the Polygon network, representing snapshots of the Bor chain state. These checkpoints are attested by a majority of the validator set before being validated and submitted on Ethereum contracts. Heimdall, an integral part of this process, manages checkpoint functionalities using the checkpoint module. It coordinates with the Bor chain to verify checkpoint root hashes when a new checkpoint is proposed. Checkpoint lifecycle and types Lifecycle Heimdall selects the next proposer using Tendermint's leader selection algorithm. The multi-stage checkpoint process is crucial due to potential failures when submitting checkpoints on the Ethereum chain caused by factors like gas limit, network traffic, or high gas fees. Each checkpoint has a validator as the proposer. The outcome of a checkpoint on the Ethereum chain (success or failure) triggers an ack (acknowledgment) or no-ack (no acknowledgment) transaction, altering the proposer for the next checkpoint on Heimdall. Flow chart representations - Checkpoint Life-cycle Flowchart - Checkpoint Module Flow Types and structures Checkpoint block header go type CheckpointBlockHeader struct { Proposer types.HeimdallAddress json:"proposer" StartBlock uint64 json:"startBlock" EndBlock uint64 json:"endBlock" RootHash types.HeimdallHash json:"rootHash" AccountRootHash types.HeimdallHash json:"accountRootHash" TimeStamp uint64 json:"timestamp" } Root hash calculation !Root Hash Image The RootHash is calculated as a Merkle hash of Bor block hashes from StartBlock to EndBlock. The process involves hashing each block's number, time, transaction hash, and receipt hash, then creating a Merkle root of these hashes. m blockHash = keccak256([number, time, tx hash, receipt hash]) Pseudocode for the root hash for 1 to n Bor blocks: go B(1) := keccak256([number, time, tx hash, receipt hash]) B(2) := keccak256([number, time, tx hash, receipt hash]) . . . B(n) := keccak256([number, time, tx hash, receipt hash]) // checkpoint is Merkle root of all block hash checkpoint's root hash = Merkel[B(1), B(2), ..., B(n)] Here are some snippets of how checkpoint is created from Bor chain block headers. Source: https://github.com/maticnetwork/heimdall/blob/develop/checkpoint/types/merkel.go#L60-L114 go // Golang representation of block data used in checkpoint blockData := crypto.Keccak256(appendBytes32(blockHeader.Number.Bytes(), new(big.Int).SetUint64(blockHeader.Time).Bytes(), blockHeader.TxHash.Bytes(), blockHeader.ReceiptHash.Bytes(),)) // array of block hashes of Bor blocks headers := [blockData1, blockData2, ..., blockDataN] // merkel tree tree := merkle.NewTreeWithOpts(merkle.TreeOptions{EnableHashSorting: false, DisableHashLeaves: true}) tree.Generate(convert(headers), sha3.NewLegacyKeccak256()) // create checkpoint's root hash rootHash := tree.Root().Hash AccountRootHash AccountRootHash is the hash of the validator account-related information that needs to pass to the Ethereum chain at each checkpoint. jsx eachAccountHash := keccak256([validator id, withdraw fee]) Pseudocode for the account root hash for 1 to n Bor blocks: go B(1) := keccak256([validator id, withdraw fee]) B(2) := keccak256([validator id, withdraw fee]) . . . B(n) := keccak256([validator id, withdraw fee]) // account root hash is Merkle root of all block hash checkpoint's account root hash = Merkel[B(1), B(2), ..., B(n)] Golang code for the account hash can be found here: https://github.com/maticnetwork/heimdall/blob/develop/types/dividend-account.go go // DividendAccount contains burned Fee amount type DividendAccount struct { User HeimdallAddress json:"user" FeeAmount string json:"feeAmount" // string representation of big.Int } // CalculateHash hashes the values of a DividendAccount func (da DividendAccount) CalculateHash() ([]byte, error) { fee, := big.NewInt(0).SetString(da.FeeAmount, 10) divAccountHash := crypto.Keccak256(appendBytes32(da.User.Bytes(), fee.Bytes(),)) return divAccountHash, nil } Messages in checkpoint module MsgCheckpoint MsgCheckpoint handles checkpoint verification on Heimdall, utilizing RLP encoding for Ethereum chain verification. It prioritizes transactions with high gas consumption to ensure only one MsgCheckpoint transaction per block. go // MsgCheckpoint represents checkpoint transaction type MsgCheckpoint struct { Proposer types.HeimdallAddress json:"proposer" StartBlock uint64 json:"startBlock" EndBlock uint64 json:"endBlock" RootHash types.HeimdallHash json:"rootHash" AccountRootHash types.HeimdallHash json:"accountRootHash" } MsgCheckpointAck MsgCheckpointAck manages successful checkpoint submissions, updating the checkpoint count and clearing the checkpointBuffer. go // MsgCheckpointAck represents checkpoint ack transaction if checkpoint is successful type MsgCheckpointAck struct { From types.HeimdallAddress json:"from" HeaderBlock uint64 json:"headerBlock" TxHash types.HeimdallHash json:"txhash" LogIndex uint64 json:"logindex" } MsgCheckpointNoAck MsgCheckpointNoAck deals with unsuccessful checkpoints or offline proposers, allowing a timeout period before selecting a new proposer. go // MsgCheckpointNoAck represents checkpoint no-ack transaction type MsgCheckpointNoAck struct { From types.HeimdallAddress json:"from" } Parameters and CLI commands Parameters The checkpoint module contains the following parameters: | Key | Type | Default value | | ------------------- | ------ | ------------------ | | CheckpointBufferTime | uint64 | 1000 time.Second | CLI commands Commands are available for various actions such as sending checkpoints, sending ack or no-ack transactions, and querying parameters. Printing all parameters go heimdallcli query checkpoint params --trust-node Expected Result: yaml checkpointbuffertime: 16m40s Send checkpoint Following command sends checkpoint transaction on Heimdall: yaml heimdallcli tx checkpoint send-checkpoint \ --start-block= \ --end-block= \ --root-hash= \ --account-root-hash= \ --chain-id= Send ack Following command sends ack transaction on Heimdall if checkpoint is successful on Ethereum: yaml heimdallcli tx checkpoint send-ack \ --tx-hash= \ --log-index= \ --header= \ --chain-id= Send no-ack Following command send no-ack transaction on Heimdall: yaml heimdallcli tx checkpoint send-noack --chain-id REST APIs Heimdall provides several REST APIs for interacting with the checkpoint module, including endpoints for preparing messages, querying checkpoints, and more. | Name | Method | Endpoint | | -- | ------ | -------------------------------- | | It returns the prepared msg for ack checkpoint | POST | /checkpoint/ack | | It returns the prepared msg for new checkpoint | POST | /checkpoint/new | | It returns the prepared msg for no-ack checkpoint | POST | /checkpoint/no-ack | | Checkpoint by number | GET | /checkpoints/ | | Get current checkpoint buffer state | GET | /checkpoints/buffer | | Get checkpoint counts | GET | /checkpoints/count | | Get last no-ack details | GET | /checkpoints/last-no-ack | | Get latest checkpoint | GET | /checkpoints/latest | | All checkpoints | GET | /checkpoints/list | | It returns the checkpoint parameters | GET | /checkpoints/parama | | It returns the prepared checkpoint | GET | /checkpoints/prepare | | Get ack count, buffer, validator set, validator count and last-no-ack details | GET | /overview | For more details and the response format of these APIs, visit Heimdall API Documentation. # governance.md: Heimdall's governance operates identically to the Cosmos SDK x/gov module, as detailed in Cosmos SDK documentation. Overview In Heimdall, *token holders can influence decisions by voting on proposals. Each token equals one vote. The governance system currently supports: - Proposal submission: Validators can submit proposals along with a deposit. If the deposit reaches the minimum threshold within a set period, the proposal moves to a voting phase. Validators can reclaim their deposits after the proposal's acceptance or rejection. - Voting: Validators are eligible to vote on proposals that have met the minimum deposit requirement. The governance module includes two critical periods: the deposit and voting periods. Proposals failing to meet the minimum deposit by the end of the deposit period are automatically rejected. Upon reaching the minimum deposit, the voting period commences, during which validators cast their votes. After the voting period, the gov/Endblocker.go script tallies the votes and determines the proposal's fate based on tallyparams: quorum, threshold, and veto. The tallying process is detailed in the source code at Heimdall GitHub repository. Types of proposals Currently, Heimdall supports param change proposals, allowing validators to modify parameters in any of Heimdall's modules. Param change proposal example For instance, validators might propose to alter the minimum txfees in the auth module. If the proposal is approved, the parameters in the Heimdall state are automatically updated without the need for an additional transaction. Command Line Interface (CLI) commands Checking governance parameters To view all parameters for the governance module: go heimdallcli query gov params --trust-node This command displays the current governance parameters, such as voting period, quorum, threshold, veto, and minimum deposit requirements. Submitting a proposal To submit a proposal: bash heimdallcli tx gov submit-proposal \ --validator-id 1 param-change proposal.json \ --chain-id proposal.json is a JSON-formatted file containing the proposal details. Querying proposals To list all proposals: go heimdallcli query gov proposals --trust-node To query a specific proposal: go heimdallcli query gov proposal 1 --trust-node Voting on a proposal To vote on a proposal: bash heimdallcli tx gov vote 1 "Yes" --validator-id 1 --chain-id Votes are automatically tallied after the voting period concludes. REST APIs Heimdall also offers REST APIs for interacting with the governance system: | Name | Method | Endpoint | | ---- | ------ | -------- | | Get all proposals | GET | /gov/proposals | | Get proposal details | GET | /gov/proposals/{proposal-id} | | Get all votes for a proposal | GET | /gov/proposals/{proposal-id}/votes | These APIs facilitate access to proposal details, voting records, and overall governance activity. # heimdall-and-bor.md: Heimdall's bor module is responsible for managing span intervals and coordinating interactions with the Bor chain. Specifically, it determines when a new span can be proposed on Heimdall based on the current block number n and the current span span. A new span proposal is permissible when the current Bor chain block number n falls within the range of span.StartBlock and span.EndBlock (inclusive of StartBlock and exclusive of EndBlock). Validators on the Heimdall chain can propose a new span when these conditions are met. Messages MsgProposeSpan The MsgProposeSpan message plays a crucial role in setting up the validator committee for a specific span and records a new span in the Heimdall state. This message is detailed in the Heimdall source code at bor/handler.go#L27. go // MsgProposeSpan creates msg propose span type MsgProposeSpan struct { ID uint64 json:"spanid" Proposer hmTypes.HeimdallAddress json:"proposer" StartBlock uint64 json:"startblock" EndBlock uint64 json:"endblock" ChainID string json:"borchainid" } Selection of producers The process for choosing producers from among all validators involves a two-step mechanism: 1. Slot allocation based on validator power: Each validator is assigned a number of slots proportional to their power. For instance, a validator with a power rating of 10 will receive 10 slots, while one with a power rating of 20 will receive 20 slots. This method ensures that validators with higher power have a correspondingly higher chance of being selected. 2. Shuffling and selection: All allocated slots are then shuffled using a seed derived from the Ethereum (ETH 1.0) block hash corresponding to each span n. The first producerCount producers are selected from this shuffled list. The bor module on Heimdall employs the Ethereum 2.0 shuffle algorithm for this selection process. The algorithm's implementation can be viewed at bor/selection.go. This method of selection ensures that the process is both fair and weighted according to the validators' power, thereby maintaining a balanced and proportional representation in the span committee. go // SelectNextProducers selects producers for the next span by converting power to slots // spanEligibleVals - all validators eligible for next span func SelectNextProducers(blkHash common.Hash, spanEligibleVals []hmTypes.Validator, producerCount uint64) (selectedIDs []uint64, err error) { if len(spanEligibleVals) <= int(producerCount) { for , val := range spanEligibleVals { selectedIDs = append(selectedIDs, uint64(val.ID)) } return } // extract seed from hash seed := helper.ToBytes32(blkHash.Bytes()[:32]) validatorIndices := convertToSlots(spanEligibleVals) selectedIDs, err = ShuffleList(validatorIndices, seed) if err != nil { return } return selectedIDs[:producerCount], nil } // converts validator power to slots func convertToSlots(vals []hmTypes.Validator) (validatorIndices []uint64) { for , val := range vals { for val.VotingPower >= types.SlotCost { validatorIndices = append(validatorIndices, uint64(val.ID)) val.VotingPower = val.VotingPower - types.SlotCost } } return validatorIndices } Types Here are the span details that Heimdall uses: go // Span structure type Span struct { ID uint64 json:"spanid" yaml:"spanid" StartBlock uint64 json:"startblock" yaml:"startblock" EndBlock uint64 json:"endblock" yaml:"endblock" ValidatorSet ValidatorSet json:"validatorset" yaml:"validatorset" SelectedProducers []Validator json:"selectedproducers" yaml:"selectedproducers" ChainID string json:"borchainid" yaml:"borchainid" } Parameters The Bor module contains the following parameters: | Key | Type | Default value | Duration () | | -------------- | ------ | ---------------------------------- | ---------------------------- | | SprintDuration. | uint64 | 16 blocks | 32 seconds | | SpanDuration | uint64 | 100 SprintDuration = 1,600 blocks | 3,200 seconds (53min and 20s) | | ProducerCount | uint64 | 4 blocks | 8 seconds | | > (): Given that blocks are produced every 2 seconds on Bor.!!! info "Sprint length" Previously, a sprint would last 64 blocks but it was agreed to decrease this number to 16 blocks on the Delhi hard fork of January 17th, 2023, precisely starting at block number 38,189,056. CLI commands Span propose tx bash heimdallcli tx bor propose-span \ --start-block \ --chain-id Query current span bash heimdallcli query bor span latest-span --chain-id Expected output: go { "spanid":2, "startblock":6656, "endblock":13055, "validatorset":{ "validators":[{ "ID":1, "startEpoch":0, "endEpoch":0, "power":1, "pubKey":"0x04b12d8b2f6e3d45a7ace12c4b2158f79b95e4c28ebe5ad54c439be9431d7fc9dc1164210bf6a5c3b8523528b931e772c86a307e8cff4b725e6b4a77d21417bf19", "signer":"0x6c468cf8c9879006e22ec4029696e005c2319c9d", "lastupdated":"", "accum":0 }], "proposer":{ "ID":1, "startEpoch":0, "endEpoch":0, "power":1, "pubKey":"0x04b12d8b2f6e3d45a7ace12c4b2158f79b95e4c28ebe5ad54c439be9431d7fc9dc1164210bf6a5c3b8523528b931e772c86a307e8cff4b725e6b4a77d21417bf19", "signer":"0x6c468cf8c9879006e22ec4029696e005c2319c9d", "lastupdated":"", "accum":0 } }, "selectedproducers":[{ "ID":1, "startEpoch":0, "endEpoch":0, "power":1, "pubKey":"0x04b12d8b2f6e3d45a7ace12c4b2158f79b95e4c28ebe5ad54c439be9431d7fc9dc1164210bf6a5c3b8523528b931e772c86a307e8cff4b725e6b4a77d21417bf19", "signer":"0x6c468cf8c9879006e22ec4029696e005c2319c9d", "lastupdated":"", "accum":0 }], "borchainid":"15001" } Query span by ID bash heimdallcli query bor span --span-id --chain-id It prints the result in same format as above. Parameters Use the following command to print all params: go heimdallcli query bor params Expected result: go sprintduration: 16 spanduration: 1600 producercount: 4 REST APIs | Name | Method | Endpoint | | ---------------- | ------ | ----------------- | | Span details | GET | /bor/span/span-id | | Get latest span | GET | /bor/latest-span | | Get params | GET | /bor/params | # introduction.md: Heimdall Heimdall is the heart of the Polygon PoS network. It performs the following functions: - Manages validators. - Handles block producer selection. - Facilitates spans. - Orchestrates the state sync mechanism between Ethereum and Polygon PoS. - Addresses other essential aspects of the system. It uses the Cosmos SDK and a forked version of Tendermint, called Peppermint. Here is the Peppermint source: https://github.com/maticnetwork/tendermint/tree/peppermint Heimdall removes certain modules from Cosmos SDK but primarily utilizes a customized version of it, following a similar pattern. # key-management.md: Each validator uses two keys to manage validator related activities on Polygon. The signer key is kept on the node and is generally considered a hot wallet, whereas the owner key is supposed to kept very secure, is used infrequently, and is generally considered a cold wallet. The staked funds are controlled by the owner key. This separation of responsibilities has been done to ensure an efficient tradeoff between security and ease of use. Both keys are Ethereum compatible addresses and work exactly the same manner. And yes, it is possible to have same owner and signer keys. Signer key The signer key is an address that is used for signing Heimdall blocks, checkpoints, and other signing related activities. This key's private key will be on the Validator node for signing purposes. It cannot manage stake, rewards or delegations. The validator must keep two types of balances on this address: - POL tokens on Heimdall (through top-up transactions) to perform validator responsibilities on Heimdall. - ETH on Ethereum chain to send checkpoints on Ethereum. Owner key The owner key is an address that is used for staking, re-stake, changing the signer key, withdraw rewards and manage delegation related parameters on the Ethereum chain. The private key for this key must be secure at all cost. All transactions through this key will be performed on the Ethereum chain. Signer change Following event is generated in case of signer change on Ethereum chain on StakingInfo.sol: https://github.com/maticnetwork/contracts/blob/develop/contracts/staking/StakingInfo.sol go // Signer change event SignerChange(uint256 indexed validatorId, address indexed oldSigner, address indexed newSigner, bytes signerPubkey); Heimdall bridge processes these events and sends transactions on Heimdall to change state based on the events. # staking.md: Staking Staking module manages validator related transactions and state for Heimdall. Note that a validator stakes their tokens on the Ethereum chain and becomes a validator. Respective validators send the transactions on Heimdall using necessary parameters to acknowledge the Ethereum stake change. Once the majority of the validators agree on the change on the stake, this module saves the validator information on Heimdall state. Messages !Stake management flow MsgValidatorJoin MsgValidatorJoin handles the staking when a new validator joins the system. Once validator calls stake or stakeFor in StakingManager.sol on Ethereum, and the new Staked event is emitted. Source: https://github.com/maticnetwork/contracts/blob/develop/contracts/staking/StakingInfo.sol#L27-L34 jsx / Staked event - emitted whenever new validator @param signer Signer address for the validator @param validatorId Validator id @param activationEpoch Activation epoch for validator @param amount Staked amount @param total Total stake @param signerPubKey Signer public key (required by Heimdall/Tendermint) / event Staked(address indexed signer, uint256 indexed validatorId, uint256 indexed activationEpoch, uint256 amount, uint256 total, bytes signerPubkey); activationEpoch is the checkpoint count from where a validator will become active on Heimdall. Stake call on smart contract fails if slots are unavailable. Validator slots are the way to restrict a number of validators in the system. Slots are managed on Ethereum smart contracts. Here is ValidatorJoin message for Heimdall transaction: go type MsgValidatorJoin struct { From hmTypes.HeimdallAddress json:"from" ID hmTypes.ValidatorID json:"id" SignerPubKey hmTypes.PubKey json:"pubkey" TxHash hmTypes.HeimdallHash json:"txhash" LogIndex uint64 json:"logindex" } MsgStakeUpdate MsgStakeUpdate handles the stake update when a validator re-stakes, or a new delegation comes in. The new StakeUpdate event is emitted in both cases. jsx / Stake update event - emitted whenever stake gets updated @param validatorId Validator id @param newAmount New staked amount / event StakeUpdate(uint256 indexed validatorId, uint256 indexed newAmount); Here is MsgStakeUpdate message for Heimdall transaction: go // MsgStakeUpdate represents stake update type MsgStakeUpdate struct { From hmTypes.HeimdallAddress json:"from" ID hmTypes.ValidatorID json:"id" TxHash hmTypes.HeimdallHash json:"txhash" LogIndex uint64 json:"logindex" } MsgValidatorExit MsgValidatorExit handles the validator exit process after a validator initiates the exit process on Ethereum. It emits the SignerUpdate event. jsx / Unstake init event - emitted whenever validator initiates the exit @param user Signer @param validatorId Validator id @param deactivationEpoch Deactivation epoch for validator @param amount Unstaked amount / event UnstakeInit(address indexed user, uint256 indexed validatorId, uint256 deactivationEpoch, uint256 indexed amount); Here is MsgValidatorExit message for Heimdall transaction: go type MsgValidatorExit struct { From hmTypes.HeimdallAddress json:"from" ID hmTypes.ValidatorID json:"id" TxHash hmTypes.HeimdallHash json:"txhash" LogIndex uint64 json:"logindex" } MsgSignerUpdate MsgSignerUpdate handles the signer update when a validator updates signer key on Ethereum. It emits SignerUpdate event. jsx / Signer change event - emitted whenever signer key changes @param validatorId Validator id @param oldSigner Current old signer @param newSigner New signer @param signerPubkey New signer public key / event SignerChange(uint256 indexed validatorId, address indexed oldSigner, address indexed newSigner, bytes signerPubkey); Here is MsgSignerUpdate message for Heimdall transaction: go // MsgSignerUpdate signer update struct type MsgSignerUpdate struct { From hmTypes.HeimdallAddress json:"from" ID hmTypes.ValidatorID json:"id" NewSignerPubKey hmTypes.PubKey json:"pubKey" TxHash hmTypes.HeimdallHash json:"txhash" LogIndex uint64 json:"logindex" } CLI Commands Validator details By signer address bash heimdallcli query staking validator-info \ --validator= \ --chain-id This command should display the following output: json { "ID":1, "startEpoch":0, "endEpoch":0, "power":10, "pubKey":"0x04b12d8b2f6e3d45a7ace12c4b2158f79b95e4c28ebe5ad54c439be9431d7fc9dc1164210bf6a5c3b8523528b931e772c86a307e8cff4b725e6b4a77d21417bf19", "signer":"0x6c468cf8c9879006e22ec4029696e005c2319c9d", "lastupdated":0, "accum":0 } By validator address bash heimdallcli query staking validator-info \ --id= \ --chain-id= This command should display the following output: json { "ID":1, "startEpoch":0, "endEpoch":0, "power":10, "pubKey":"0x04b12d8b2f6e3d45a7ace12c4b2158f79b95e4c28ebe5ad54c439be9431d7fc9dc1164210bf6a5c3b8523528b931e772c86a307e8cff4b725e6b4a77d21417bf19", "signer":"0x6c468cf8c9879006e22ec4029696e005c2319c9d", "lastupdated":0, "accum":0 } Validator join This command sends validator-join command through CLI: bash heimdallcli tx staking validator-join \ --signer-pubkey \ --tx-hash \ --log-index \ --chain-id tx-hash value must be the same as Ethereum transaction hash which emitted Staked event, and log-index must be the same as the index at which the event is emitted. REST APIs | Name | Method | Endpoint | | -------------------------- | ------ | ---------------------------------- | | Get Heimdall validator set | GET | /staking/validator-set | | Get validator details | GET | /staking/validator/validator-id | All query APIs will result in following format: json { "height": "1", "result": { ... } } # topup.md: Top-ups are amounts used to pay fees on the Heimdall chain. There are two ways to top up your account: 1. When new validator joins, they can mention a topup amount in addition to the staked amount, which will be moved as balance on Heimdall chain to pays fees on Heimdall. 2. A user can directly call the top-up function on the staking smart contract on Ethereum to increase top-up balance on Heimdall. Messages MsgTopup MsgTopup transaction is responsible for minting balance to an address on Heimdall based on Ethereum chain's TopUpEvent on staking manager contract. Handler for this transaction processes top-up and increases the balance only once for any given msg.TxHash and msg.LogIndex. It throws Older invalid tx found error, if trying to process the top-up more than once. Here is the structure for the top-up transaction message: go type MsgTopup struct { FromAddress types.HeimdallAddress json:"fromaddress" ID types.ValidatorID json:"id" TxHash types.HeimdallHash json:"txhash" LogIndex uint64 json:"logindex" } MsgWithdrawFee MsgWithdrawFee transaction is responsible for withdrawing balance from Heimdall to Ethereum chain. A validator can withdraw any amount from Heimdall. Handler processes the withdraw by deducting the balance from the given validator and prepares the state to send the next checkpoint. The next possible checkpoint will contain the withdraw related state for the specific validator. Handler fetches validator information based on the ValidatorAddress and processes the withdrawal. go // MsgWithdrawFee - high-level transaction of the fee coin withdrawal module type MsgWithdrawFee struct { ValidatorAddress types.HeimdallAddress json:"fromaddress" Amount types.Int json:"amount" } CLI commands Top-up fee bash heimdallcli tx topup fee --log-index --tx-hash --validator-id --chain-id Withdraw fee bash heimdallcli tx topup withdraw --chain-id To check the updated top-up amount for the account, run the following command: bash heimdallcli query auth account --trust-node REST APIs | Name | Method | URL | Body Params | | ------------ | ------ | ------------- | -- | | Top-up fee | POST | /topup/fee | id Validator ID, txhash Transaction hash of successful top-up event on Ethereum chain, logindex Log index of the top-up event emitted on Ethereum | | Withdraw fee | POST | /topup/withdraw | amount Withdraw amount | # validation.md: Heimdall's ante handler plays a crucial role in the integrity and efficiency of transaction processing. It is primarily responsible for the preliminary verification and validation of all transactions, ensuring that they meet the necessary criteria before being included in a block. This includes checking the sender's balance to ensure there are sufficient funds to cover transaction fees and subsequently deducting these fees for successful transactions. Advanced gas management in Heimdall Block and transaction gas limits Heimdall employs a gas limit system to regulate the computational and storage resources consumed by transactions and blocks. This system is designed to prevent excessive block sizes and ensure network stability. Block gas limit Each block in Heimdall has a maximum gas limit, constraining the total gas used by all transactions within the block. The sum of the gas used by each transaction in a block must not exceed this limit: go block.GasLimit >= sum(tx1.GasUsed + tx2.GasUsed + ... + txN.GasUsed) The maximum block gas limit and block size are specified as part of the consensus parameters during the application setup, as seen in the Heimdall source code at app.go#L464-L471: go maxGasPerBlock int64 = 10000000 // 10 Million maxBytesPerBlock int64 = 22020096 // 21 MB // Setting consensus parameters ConsensusParams: &abci.ConsensusParams{ Block: &abci.BlockParams{ MaxBytes: maxBytesPerBlock, MaxGas: maxGasPerBlock, }, ... }, Transaction gas limit For individual transactions, the gas limit is determined by parameters in the auth module and can be modified through Heimdall's governance (gov) module. Special handling of checkpoint transactions Checkpoint transactions, which require Merkle proof verification on the Ethereum chain, are treated distinctly. To streamline processing and avoid the overhead of additional Merkle proof verification, Heimdall restricts blocks containing a MsgCheckpoint transaction to just that one transaction: go // Gas requirement for checkpoint transaction gasWantedPerCheckpoinTx sdk.Gas = 10000000 // 10 Million // Special gas limit handling for checkpoint transactions if stdTx.Msg.Type() == "checkpoint" && stdTx.Msg.Route() == "checkpoint" { gasForTx = gasWantedPerCheckpoinTx } Enhanced transaction verification and replay protection The ante handler in Heimdall is instrumental in ensuring the legitimacy and uniqueness of transactions. It performs a thorough verification of incoming transactions, including signature validation, as delineated in the source code at ante.go#L230-L266. Sequence number for replay protection A critical aspect of transaction security in Heimdall is the use of a sequenceNumber in each transaction. This feature is a safeguard against replay attacks, where a transaction might be fraudulently or mistakenly repeated. To prevent such scenarios, the ante handler increments the sequence number for the sender's account after each successful transaction. This incrementation ensures that each transaction is unique and that previous transactions cannot be replayed. In summary, Heimdall's ante handler, along with its sophisticated gas management and transaction verification systems, provides a robust framework for secure and efficient transaction processing. The careful balance of block and transaction gas limits, coupled with advanced replay protection mechanisms, ensures the smooth operation of the Heimdall chain within the Polygon PoS network. # matic.md: !!! info "Transitioning to POL" Polygon network is transitioning from MATIC to POL, which will serve as the gas and staking token on Polygon PoS. Use the links below to learn more: - Migrate from MATIC to POL - POL token specs MATIC is a native cryptocurrency that served as the gas token on Polygon PoS, analogous to how ETH functions within the Ethereum ecosystem. To engage with the Polygon PoS network, users previously used MATIC tokens to cover gas fees for transactions, smart contract interactions, and as a part of the network's security model via staking. This is an example script to send MATIC tokens from one account to another on the Polygon chain. !!! info "MATIC rewards" Validator and staking rewards will now be distributed in the form of POL tokens. You can obtain test MATIC for testing purposes on the Amoy testnet. Testnet tokens don't hold any real-world value. This can be done via the Polygon faucet in the following way: 1. Visit the Polygon Faucet. 2. Select MATIC Token. 3. Choose the Polygon PoS (Amoy) network. 4. Enter your account address and confirm. For interactions between Sepolia and Amoy, consider using tools like Matic.js, or use the Polygon Portal. # pol.md: POL is the native token upgrade for the Polygon ecosystem, designed for use in a wide range of activities and various purposes. For instance, POL is used as a tool for network participation and security. By staking POL, participants can actively contribute to the ecosystem as validators. Importantly, like MATIC, POL is built on OpenZeppelin's ERC20 implementations, supports EIP-2612 for signature-based permit approvals, and thus inherits most of the features found in MATIC. PIPs Community-driven governance and feedback played a crucial role in refining the POL token's design and functionality. Learn more about the proposals by following the links below. 1. PIP-17: Polygon Ecosystem Token (POL) 2. PIP-18: Polygon 2.0 Phase 0 - Frontier 3. PIP-19: Update Polygon PoS Native Token to POL 4. PIP-25: Adjust POL Total Supply 5. PIP-26: Transition from MATIC to POL Validator Rewards !!! info "Initial amount of POL tokens" In the case of POL, the initial amount (the total number of POL when the upgrade occurs) is 10 billion tokens â€" 1:1 with MATIC since this is an upgrade. Do I need to do anything manually? Please refer to the MATIC to POL migration guide for details on what action you need to take, if any,* depending on the chain they are currently located, i.e., Ethereum, Polygon PoS, Polygon zkEVM, etc. !!! tip "Deep dive into POL" Read the detailed blog post on the POL migration to learn more about the POL token, its properties, and what the migration means for the Polygon ecosystem. As a dApp developer, feel free to review the PIP's to analyze the changes to the token protocol. Ideally, developers shouldn't see any breaking changes. But if you do, feel free to reach out to us via the Polygon R&D Discord. POL technical information Does the amount of POL increase over time? Yes, the amount of POL will increase on an emissions schedule that has reached community consensus. Originally proposed as a 2% yearly emission rate, with 1% to the community treasury and 1% to validator rewards, community consensus was reached in PIP-26 to continue the original emissions reward schedule, in addition to the proposed POL emissions rate, with an end date of the original emissions reward schedule in June 2025. PIP-26 revised the percentage of validator rewards to 2% for the fourth year (2023-2024), 1.5% for the fifth year (2024-2025), and 1% thereafterâ€"for an effective emission of 2% increase of POL per year beginning after June 2025. Governance may change this rate through an upgrade of the EmissionManager contract. How is POL minted? The EmissionManager smart contract is responsible for initiating the upgrade to POL through a minting process. This contract is upgradeable, allowing for future changes through governance. It also ensures that the StakeManager and Treasury contracts receive their respective amounts of the newly minted tokens. What determines the emission rate? The emission rate is governed by a variable named mintPerSecondCap in the primary POL smart contract. Additionally, the EmissionManager contract uses a constant called INTERESTPERYEARLOG2 to calculate an annual emission rate, compounded per year. Can the emission rate be modified? Yes, the emission rate can be modified through a governance proposal, but cannot surpass mintPerSecondCap in the primary POL smart contract. What considerations go into POLâ€™s design? The economic design of POL incorporates several key considerations to aim for stability, such as: - Community Governance: Active community participation in governance processes allows for adaptability and responsiveness to changing conditions. - Smart Contract Security: The integrity of the underlying smart contracts is crucial for maintaining a stable environment. Token migration and reversal What is the purpose of token migration? Token migration serves the purpose of allowing for the upgrade from MATIC to POL. This migration operates on a 1-to-1 conversion basis. A migration smart contract will allow users to upgrade from MATIC to POL by calling a smart contract function that will accept MATIC and provide an equal amount of POL in return. The contract is designed to permit the entire supply of MATIC tokens to be upgraded. What happens to the MATIC tokens after migration? MATIC is held in the migration contract and can be used for unmigration. Can POL tokens be reverted back to MATIC? Yes, the migration contract includes a feature known as 'unmigration'. This allows users to convert their POL to an equivalent amount of MATIC. Governance controls this feature, providing flexibility in response to network conditions or security concerns. Bridging mechanisms How does the modified bridge function? The bridge will undergo modifications, with community approval, to change the native token of Polygon PoS to the new POL token. Specifically, the following changes are being proposed: - Bridging POL to Polygon PoS: if you bridge POL tokens to Polygon PoS, you will receive an equal amount of native tokens (POL) on Polygon PoS. - Bridging POL to Ethereum: when bridging native tokens (POL), the bridge will always disburse POL tokens. Are there any breaking changes? Yes, if an existing contract relies on receiving MATIC from a bridge and receives POL instead, this might result in locked funds. Developers must check their contracts, verify the transaction lifecycle, and engage on the forum for any doubts. Governance and security protocols Who holds the authority to govern the POL-based contracts? The contracts are governed by the Polygon decentralized governance model. In accordance with the PIP process, the community can propose changes and provide feedback. What security measures are in place? The contracts have been designed with various security measures, including rate limits on minting and the ability to lock or unlock features like 'unmigration'. Implications and safeguards How is POL used to reward ecosystem participation? POL's design aims to foster a sustainable and predictable growth model. This model primarily rewards active contributors and participants within the ecosystem. How can I avoid scams? Always verify contract addresses and use reputable platforms for transactions. Exercise extreme caution when dealing with claims like "swaps" or â€œtransfersâ€ from unverified sources. # eip-1559.md: The London hard fork introduced a new EIP that modifies how gas estimation and costs work for transactions on Polygon. Due to this, there is a change in how the transaction object is formed when sending transactions on Polygon. A new transaction type called Type 2 Transaction has been introduced. The legacy type transactions will still be compatible but it is recommended to shift to the new style. You can navigate to the end of this document to directly peek into the code. How legacy transactions (Type 0) work When you submit a transaction, you also send a gasPrice which is an amount you are offering to pay per gas consumed. Then, when you submit the transaction, miners can decide to include your transaction or not based on your gasPrice bid. Miners will prioritize the highest gas prices. Sending Type 2 transactions with EIP1559 It is a similar concept, the gasPrice will be split into a baseFee and a priorityFee. Every transaction needs to pay the base fee, which is calculated based on how full the previous block was. Transactions can also offer the miner a priorityFee to incentivize the miner to include the transaction in the block. Sending legacy transactions Only the gasPrice needed to be mentioned in the legacy transaction prior to the London fork. The following code example shows sending transaction using a type 0 transaction: jsx const sendLegacyTransaction = async () => { const web3 = new Web3('https://polygon-rpc.com'); await web3.eth.sendTransactions({ from: 0x05158d7a59FA8AC5007B3C8BabAa216568Fd32B3, to: 0xD7Fbe63Db5201f71482Fa47ecC4Be5e5B125eF07, value: 1000000000000000000, gasPrice: 200000000000 }) } Sending EIP1559 transactions Add maxPriorityFeePerGas field The closest analogy to the gas:gasPrice combination is gas:maxPriorityFeePerGas. Since the baseFee needs to be paid regardless, we can just submit a bid on the tip for the miner. Note that the Polygon Gas Station V2 can be used to get the gas fee estimates. The following code example shows sending transaction in Type 2 method: jsx // Example for const sendEIP1559Transaction = async () => { const web3 = new Web3('https://polygon-rpc.com'); await web3.eth.sendTransactions({ from: 0xFd71Dc9721d9ddCF0480A582927c3dCd42f3064C, to: 0x8C400f640447A5Fc61BFf7FdcE00eCf20b85CcAd, value: 1000000000000000000, maxPriorityFeePerGas: 40000000000 }) } The Polygon Gas Station V2 can be used to get the gas fee estimates. Polygon Gas Station V2 Endpoint: jsx https://gasstation.polygon.technology/v2 Polygon Gas Station V2 sample response: jsx { "safeLow": { "maxPriorityFee": 37.181444553750005, "maxFee": 326.2556979087 }, "standard": { "maxPriorityFee": 49.575259405, "maxFee": 435.00759721159994 }, "fast": { "maxPriorityFee": 61.96907425625, "maxFee": 543.7594965144999 }, "estimatedBaseFee": 275.308812719, "blockTime": 6, "blockNumber": 23948420 } See also Please read the following articles to get a better understanding of sending EIP-1559 transactions: How to send transactions with EIP 1559, this tutorial will walk you through both the legacy and new (EIP-1559) way to estimate gas and send transactions. Learn how to send an EIP-1559 transaction using ethers.js. # eip-4337.md: The ERC-4337 standard, also known as EIP-4337, allows developers to achieve account abstraction on the Polygon PoS. This page provides a simplified overview of the different components of ERC-4337 and how they work together. The ERC-4337 standard consists of four main components: UserOperation, Bundler, EntryPoint, and Contract Account. Optional components include Paymasters and Aggregators. Building ERC-4337 transactions ERC-4337 transactions are called UserOperations to avoid confusion with the regular transaction type. UserOperations are pseudo-transaction objects that are used to execute transactions with contract accounts. ERC-4337 transactions have to be sent to nodes that include ERC-4337 bundlers. The UserOperation object has a few fields. | Field | Type | Description | | ---------------------- | --------- | --- | | sender | address | The address of the smart contract account | | nonce | uint256 | Anti-replay protection | | initCode | bytes | Code used to deploy the account if not yet on-chain | | callData | bytes | Data that's passed to the sender for execution | | callGasLimit | uint256 | Gas limit for execution phase | | verificationGasLimit | uint256 | Gas limit for verification phase | | preVerificationGas | uint256 | Gas to compensate the bundler | | maxFeePerGas | uint256 | Similar to EIP-1559 max fee | | maxPriorityFeePerGas | uint256 | Similar to EIP-1559 priority fee | | paymasterAndData | bytes | Paymaster Contract address and any extra data required for verification and execution | | signature | bytes | Used to validate a UserOperation along with the nonce during verification | ERC-4337 wallets Users must have an ERC-4337 smart contract account to validate UserOperations. The core interface for an ERC-4337 wallet is: solidity interface IAccount { function validateUserOp (UserOperation calldata userOp, bytes32 userOpHash, address aggregator, uint256 missingAccountFunds) external returns (uint256 sigTimeRange); } Resources - ERC-4337 proposal is the link to the official proposal and technical specification. - @account-abstraction SDK is an npm package for using ERC-4337 developed by the authors of the proposal. - Stackup provides node services with ERC-4337 bundlers and other ERC-4337 infrastructure. - WalletKit is an all-in-one platform for adding smart, gasless wallets to your app. It has integrated support for ERC-4337 and comes with a paymaster and bundler included, requiring no extra setup. # meta-transactions.md: The current state of transacting The traditional transaction model on Ethereum and similar blockchains has notable limitations. One key issue is that users must pay gas fees to initiate transactions, which can be a barrier, as they often need to acquire cryptocurrency first. To address this, the transaction sender can be decoupled from the gas payer. This allows for scaling transaction execution and creating a more seamless user experience. By implementing middleware through a third party, gas payments can be handled separately. This is where meta transactions come in. What are meta transactions? Meta transactions enable users to interact with the blockchain without needing tokens to cover transaction fees. This is achieved by decoupling the transaction sender from the gas payer. In this model, the executor submits a transaction request by signing the intended action with their private key, rather than paying gas directly. The meta transaction consists of a standard Ethereum transaction augmented with additional parameters. The signed transaction parameters are sent to a secondary network acting as a relayer. Relayers validate transactions based on relevance to the dApp, then wrap the request into a standard transaction, paying the gas fee. The network broadcasts this transaction, and the contract unwraps it by validating the original signature, executing the action on behalf of the user. !!! info "Meta transactions vs. batch transactions" To clarify: a meta transaction is different from a batch transaction, where a batch transaction is a transaction that can send multiple transactions at once and are then executed from a single sender (single nonce specified) in sequence. In summary, meta transactions are a design pattern where: - A user (sender) signs a request with their private key and sends it to a relayer - The relayer wraps the request into a transaction and sends it to a contract - The contract unwraps the transaction and executes it Native transactions imply that the sender is also the payer. When taking the payer away from the sender, the sender becomes more like an intender - the sender shows the intent of the transaction they would like executed on the blockchain by signing a message containing specific parameters related to their message, and not an entirely constructed transaction. Use cases One can imagine the capabilities of meta transactions for scaling dApps and interactions with smart contracts. Not only can a user create a gasless transaction, but they can also do so many times, and with an automation tool, meta transactions can influence the next wave of applications for practical use cases. Meta transactions enable real utility in smart contract logic, which is often limited because of gas fees and the interactions required on-chain. Let's look at a few scenarios highlighting how meta transactions can enhance user experience in dApps. Voting A user wishing to participate in on-chain governance can vote through a voting contract by signing a message with their decision. Traditionally, they would need to pay gas fees and know how to interact with the contract directly. But in this case they sign a meta transaction containing the vote details off-chain and send it to a relayer. The relayer receives the signed message, validates the priority of the vote, wraps it into a standard transaction, pays the gas fees, and submits it to the voting contract. Once validated, the contract executes the vote on behalf of the user. Gaming In blockchain-based games, players often need to pay gas fees to perform in-game actions like trading items or upgrading characters. By using meta transactions, players can interact with the game without needing to hold ETH, making it easier for casual gamers to enjoy the experience without the hassle of managing crypto. Minting NFTs In NFT marketplaces, users often face high gas fees when minting, buying, or selling NFTs. By utilizing meta transactions, users can create or purchase NFTs without having to manage Ether for gas. They simply sign the transaction request, and a relayer submits it on their behalf, enhancing user experience and lowering the barrier to entry for those unfamiliar with handling cryptocurrencies. Try 'em out Assuming your familiarity with the different approaches you can take to integrate meta transactions in your dApp, and depending on whether you're migrating to meta transactions or building fresh dApp on using it. To integrate your dApp with meta transactions on Polygon PoS, you can choose to go with one of the following relayers or spin up a custom solution: - Biconomy - Gas Station Network (GSN) - Infura - Gelato Goal Execute transactions on Polygon PoS without changing provider on MetaMask (this tutorial caters to MetaMask's in-page provider, can be modified to execute transactions from any other provider) Under the hood, user signs on an intent to execute a transaction, which is relayed by a simple relayer to execute it on a contract deployed on Polygon chain. What is enabling transaction execution? The client that the user interacts with (web browser, mobile apps, etc.) never interacts with the blockchain, instead it interacts with a simple relayer server (or a network of relayers), similar to the way GSN or any meta-transaction solution works. For any action that requires blockchain interaction, - Client requests an EIP-712 formatted signature from the user - The signature is sent to a simple relayer server (should have a simple auth/spam protection if used for production, or Biconomy's Mexa SDK can be used: https://github.com/bcnmy/mexa-sdk) - The relayer interacts with the blockchain to submit user's signature to the contract. A function on the contract called executeMetaTransaction processes the signature and executes the requested transaction (via an internal call). - The relayer pays for the gas making the transaction effectively free ðŸ¤‘ Example implementation - Choose between a custom simple relayer node/Biconomy. - For Biconomy, setup a dApp from the dashboard and save the api-id and api-key, see: https://docs.biconomy.io/ Steps: 1. Let's register our contracts to Biconomy dashboard 1. Visit Biconomy's offical docs. 2. Navigate and login to the Dashboard. 3. Select Polygon Amoy Testnet when registering your dApp. 2. Copy the API key to use for you dApp's frontend. 3. And add function executeMetaTransaction in Manage-Api and make sure to enable meta-tx (Check native-metatx option). - If you'd like to use your own custom API that sends signed transactions on the blockchain, you can refer to the server code here: https://github.com/angelagilhotra/ETHOnline-Workshop/tree/master/2-network-agnostic-transfer - Make sure that the contract you'd like to interact with inherits from NativeMetaTransactions - ðŸ'€ peep into executeMetaTransaction function in the contract. - Link: https://github.com/maticnetwork/pos-portal/blob/34be03cfd227c25b49c5791ffba6a4ffc9b76036/flat/ChildERC20.sol#L1338 jsx let data = await web3.eth.abi.encodeFunctionCall({ name: 'getNonce', type: 'function', inputs: [{ name: "user", type: "address" }] }, [accounts[0]]); let nonce = await web3.eth.call ({ to: token["80001"], data }); const dataToSign = getTypedData({ name: token["name"], version: '1', salt: '0x0000000000000000000000000000000000000000000000000000000000013881', verifyingContract: token["80001"], nonce: parseInt(nonce), from: accounts[0], functionSignature: functionSig }); const msgParams = [accounts[0], JSON.stringify(dataToSign)]; let sig = await eth.request ({ method: 'ethsignTypedDatav3', params: msgParams }); - Once you have a relayer and the contracts set up, what is required is for the client to be able to fetch an EIP-712 formatted signature and simply call the API with the required parameters ref: https://github.com/angelagilhotra/ETHOnline-Workshop/blob/6b615b8a4ef00553c17729c721572529303c8e1b/2-network-agnostic-transfer/sign.js#L47 jsx let data = await web3.eth.abi.encodeFunctionCall({ name: 'getNonce', type: 'function', inputs: [{ name: "user", type: "address" }] }, [accounts[0]]); let nonce = await web3.eth.call ({ to: token["80001"], data }); const dataToSign = getTypedData({ name: token["name"], version: '1', salt: '0x0000000000000000000000000000000000000000000000000000000000013881', verifyingContract: token["80001"], nonce: parseInt(nonce), from: accounts[0], functionSignature: functionSig }); const msgParams = [accounts[0], JSON.stringify(dataToSign)]; let sig = await eth.request ({ method: 'ethsignTypedDatav3', params: msgParams }); Calling the API, ref: https://github.com/angelagilhotra/ETHOnline-Workshop/blob/6b615b8a4ef00553c17729c721572529303c8e1b/2-network-agnostic-transfer/sign.js#L110 jsx const response = await request.post('http://localhost:3000/exec', { json: txObj, }, (error, res, body) => { if (error) { console.error(error) return } document.getElementById(el).innerHTML = response:+ JSON.stringify(body) }) If using Biconomy, the following should be called: jsx const response = await request.post('https://api.biconomy.io/api/v2/meta-tx/native', { json: txObj, }, (error, res, body) => { if (error) { console.error(error) return } document.getElementById(el).innerHTML = response:+ JSON.stringify(body) }) where the txObj should look something like: json { "to": "0x2395d740789d8C27C139C62d1aF786c77c9a1Ef1", "apiId": , "params": ["0x2173fdd5427c99357ba0dd5e34c964b08079a695", "0x2e1a7d4d000000000000000000000000000000000000000000000000000000000000000a", "0x42da8b5ac3f1c5c35c3eb38d639a780ec973744f11ff75b81bbf916300411602", "0x32bf1451a3e999b57822bc1a9b8bfdfeb0da59aa330c247e4befafa997a11de9", "27"], "from": "0x2173fdd5427c99357ba0dd5e34c964b08079a695" } - If you use the custom API, it executes the executeMetaTransaction function on the contract: (ref: https://github.com/angelagilhotra/ETHOnline-Workshop/blob/6b615b8a4ef00553c17729c721572529303c8e1b/2-network-agnostic-transfer/server/index.js#L40) jsx try { let tx = await contract.methods.executeMetaTransaction(txDetails.from, txDetails.fnSig, r, s, v).send ({ from: user, gas: 800000 }) req.txHash = tx.transactionHash } catch (err) { console.log (err) next(err) } is using biconomy, the client side call looks like: jsx // client/src/App.js import React from "react"; import Biconomy from "@biconomy/mexa"; const getWeb3 = new Web3(biconomy); biconomy .onEvent(biconomy.READY, () => { // Initialize your dapp here like getting user accounts etc console.log("Mexa is Ready"); }) .onEvent(biconomy.ERROR, (error, message) => { // Handle error while initializing mexa console.error(error); }); / use the getWeb3 object to define a contract and calling the function directly / # becoming-a-validator.md: !!! info "Transitioning to POL" Polygon network is transitioning from MATIC to POL, which will serve as the gas and staking token on Polygon PoS. Use the links below to learn more: - Migrate from MATIC to POL - POL token specs Validators are the key actor in maintaining the Polygon PoS network. Validators run a full node, secure the network by staking POL to produce blocks, validate and participate in PoS consensus. Validators play a crucial role in maintaining the integrity and security of the Polygon PoS network. By running a full node, validators contribute to the network's infrastructure and help facilitate transactions. They secure the network by staking POL tokens, which enables them to produce blocks and participate in network consensus (PoS). This contributes to greater network security and enhances network decentralization making it more resilient against potential attacks. In return for their efforts, validators are rewarded with incentives, encouraging active participation and commitment to the network's stability and growth. !!! info "Active validator limit" There is limited space for accepting new validators. Currently, the system allows for a maximum of 105 active validators at any given time. New validators can join the active set only when a currently active validator unbonds or is removed due to low performance. If you are interested in becoming a validator on the PoS network, you can submit an application over at the Polygon validators hub. \ Submitting an application does not guarantee a validator slot. Overview Polygon consists of the three following layers: Ethereum layer: A set of contracts on the Ethereum mainnet. Heimdall layer: A set of proof-of-stake Heimdall nodes running in parallel to the Ethereum mainnet, monitoring the set of staking contracts deployed on the Ethereum mainnet, and committing the Polygon PoS network checkpoints to the Ethereum mainnet. Heimdall is based on Tendermint. Bor layer: A set of block-producing Bor nodes shuffled by Heimdall nodes. Bor is based on Go Ethereum. To be a validator on the Polygon PoS network, you must do the following: Run a sentry node: A separate machine running a Heimdall node and a Bor node. A sentry node is open to all nodes on the Polygon PoS network. Run a validator node: A separate machine running a Heimdall node and a Bor node. A validator node is only open to its sentry node and closed to the rest of the network. Stake the POL tokens in the staking contracts deployed on the Ethereum mainnet. Components Heimdall Heimdall does the following: Monitors the staking contracts on the Ethereum mainnet. Verifies all state transitions on the Bor chain. Commits the Bor chain state checkpoints to the Ethereum mainnet. Heimdall is based on Tendermint. !!! info "See also" GitHub repository: Heimdall GitHub repository: Staking contracts Blog post: Heimdall and Bor Bor Bor does the following: Produces blocks on Polygon PoS. Bor is the block producing node and layer for the Polygon PoS network. It is based on Go Ethereum. Blocks produced on Bor are validated by Heimdall nodes. !!! info "See also" GitHub repository: Bor Blog post: Heimdall and Bor Validator responsibilities !!! tip "Stay in the know" Keep up with the latest node and validator updates from the Polygon team and the community by keeping an eye on the announcements posed to Polygon forums. A blockchain validator is someone who is responsible for validating transactions within a blockchain. On the Polygon PoS network, any participant can be qualified to become a Polygon's validator by running a validator node (sentry + validator) to earn rewards and collect transaction fees. To ensure the good participation by validators, they lock up at least 1 POL token as a stake in the ecosystem. !!! info "PIP4 raised the minimum staking amount" After the implementation of the PIP4 governance proposal at the contract level, the minimum staking amount was increased to 10,000 POL. Any validator on the Polygon PoS network has the following responsibilities: - Technical node operations (done automatically by the nodes). - Operations - Maintain high uptime. - Check node-related services and processes daily. - Run node monitoring. - Keep ETH balance (between 0.5 to 1) on the signer address. - Delegation - Be open to delegation. - Communicate commission rates. - Communication - Communicate issues. - Provide feedback and suggestions. - Earn staking rewards for validate blocks on the blockchain. Technical node operations The following technical node operations are done automatically by the nodes: Block producer selection: Select a subset of validators for the block producer set for each span. For each span, select the block producer set again on Heimdall and transmit the selection information to Bor periodically. Validating blocks on Bor: For a set of Bor blocks, each validator independently reads block data for these blocks and validates the data on Heimdall. Checkpoint submission: A proposer is chosen among the validators for each Heimdall block. The checkpoint proposer creates the checkpoint of Bor block data, validates, and broadcasts the signed transaction for other validators to consent to. If more than 2/3 of the active validators reach consensus on the checkpoint, the checkpoint is submitted to the Ethereum mainnet. Sync changes to Polygon staking contracts on Ethereum: Continuing from the checkpoint submission step, since this is an external network call, the checkpoint transaction on Ethereum may or may not be confirmed, or may be pending due to Ethereum congestion issues. In this case, there is an ack/no-ack process that is followed to ensure that the next checkpoint contains a snapshot of the previous Bor blocks as well. For example, if checkpoint 1 is for Bor blocks 1-256, and it failed for some reason, the next checkpoint 2 will be for Bor blocks 1-512. See also Heimdall architecture: Checkpoint. State sync from the Ethereum mainnet to Bor: Contract state can be moved between Ethereum and Polygon, specifically through Bor. A dApp contract on Ethereum calls a function on a special Polygon contract on Ethereum. The corresponding event is relayed to Heimdall and then Bor. A state-sync transaction gets called on a Polygon smart contract and the dApp can get the value on Bor via a function call on Bor itself. A similar mechanism is in place for sending state from Polygon to Ethereum. See also State Sync Mechanism. Operations Maintain high uptime The node uptime on the Polygon PoS network is based on the number of checkpoint transactions that the validator node has signed. Approximately every 34 minutes a proposer submits a checkpoint transaction to the Ethereum mainnet. The checkpoint transaction must be signed by every validator on the Polygon PoS network. Failure to sign a checkpoint transaction results in the decrease of your validator node performance. The process of signing the checkpoint transactions is automated. To ensure your validator node is signing all valid checkpoint transactions, you must maintain and monitor your node health. Check node services and processes daily You must check daily the services and processes associated with Heimdall and Bor. Also, pruning of the nodes should be done regularly to reduce disk usage. Run node monitoring You must run either: Grafana Dashboards provided by Polygon. See GitHub repository: Matic-Jagar setup Or, use your own monitoring tools for the validator and sentry nodes. Ethereum endpoint used on nodes should be monitored to ensure the node is within the request limits. Maintain ETH balance You must maintain an adequate amount of ETH (should be always around the threshold value i.e., 0.5 to 1) on your validator signer address on the Ethereum Mainnet. You need ETH to: Sign the proposed checkpoint transactions on the Ethereum Mainnet. Propose and send checkpoint transactions on the Ethereum Mainnet. Not maintaining an adequate amount of ETH on the signer address will result in: Delays in the checkpoint submission. Note that transaction gas prices on the Ethereum network may fluctuate and spike. Delays in the finality of transactions included in the checkpoints. Delays in subsequent checkpoint transactions. Delegation Be open for delegation All validators must be open for delegation from the community. Each validator has the choice of setting their own commission rate. There is no upper limit to the commission rate. Communicate commission rates It is the moral duty of the validators to communicate the commission rates and the commission rate changes to the community. The preferred platforms to communicate the commission rates are: Discord Forum Communication Communicate issues Communicating issues as early as possible ensures that the community and the Polygon team can rectify the problems as soon as possible. The preferred platforms to communicate the commission rates are: Discord Forum GitHub Provide feedback and suggestions At Polygon, we value your feedback and suggestions on any aspect of the validator ecosystem. Forum is the preferred platform to provide feedback and suggestions. Run and maintain a node The following step-by-step guides will take you through the process of running a new validator node, or performing necessary maintenance actions for an existing node you've deployed. Join the network as a validator Start and run the nodes with Ansible. Start and run the nodes with binaries. Stake as a validator. Maintain your validator nodes Change the signer address. Change the commission. Community assistance Discord Forum # building-on-polygon.md: !!! info "Transitioning to POL" Polygon network is transitioning from MATIC to POL, which will serve as the gas and staking token on Polygon PoS. Use the links below to learn more: - Migrate from MATIC to POL - POL token specs Overview All your favorite Ethereum tools (Foundry, Remix, Web3.js) work seamlessly on Polygon, with the same familiar UX. Just switch to the Polygon RPC and keep building. Connect your wallet and deploy any decentralized application to either PoS mainnet or Amoy testnet (Sepolia-anchored). Use the links below to find the right tooling and guides that suit your needs the best. - Faucets - Fetch test tokens - Polygon gas station - Gas estimation API - Polygon dApp Launchpad - dApp development CLI tool - Popular third-party tooling - Matic.js library If you have no prior experience in dApp development, the following resources will help you get started with some essential tools for building, testing, and deploying applications on Polygon PoS. - Full Stack dApp: Tutorial Series - Web3.js - Ethers.js - thirdweb - Remix - Hardhat - Foundry - Metamask - Venly (previously Arkane) - Develop a dApp using Fauna, Polygon, and React Network details To access network-related details, including the chain ID, RPC URL, and more, for both the mainnet and Amoy testnet, refer to the network documentation. Wallets You'll need an Ethereum-based wallet to interact with Polygon because the network runs on Ethereum Virtual Machine (EVM). You can choose to set up aÂ MetaMask, orÂ Venly (Arkane)Â Wallet. There are several other third-party wallet options available to choose from, and you'll find them listed out on this page here. !!! tip "Set up web3 provider" Refer to the following guides and follow along to set up your wallet for making web3 function calls: - Venly Common tasks Token bridging between Polygon PoS and Ethereum and vice-versa, and inter-layer communication are basic and essential actions that most dApps need to perform. Use the links below to navigate to guides that'll help you get started with these tasks. Bridge tokens from Ethereum to PoS Bridge tokens from PoS to Ethereum L1 - L2 communication and state transfer Connecting to Polygon You can add Polygon to MetaMask, or directly use Venly (Arkane), which allows you to connect to Polygon usingÂ RPC. In order to connect with the Polygon PoS network to read blockchain information, you can use a node provider like Alchemy SDK. js // Javascript // Setup: npm install alchemy-sdk const { Alchemy, Network } = require("alchemy-sdk"); const settings = { apiKey: "demo", // Replace with your API Key from https://www.alchemy.com network: Network.MATICMAINNET, // Replace with MATICAMOY for testnet config }; const alchemy = new Alchemy(settings); async function main() { const latestBlock = await alchemy.core.getBlockNumber(); console.log("The latest block number is", latestBlock); } main(); !!! tip "Reach out to us!" If you're encountering problems while hacking or have questions about something, please use the following methods to contact us: 1. If you come across a complex code repository, 404s on the docs site, or if you feel there's something missing - feel free to open an issue on the Polygon Knowledge Layer's GitHub repository. You can also open a PR if you're looking to contribute! 2. Get in touch with us via Discord: - Community Discord - Research and Development Discord Already have a dApp? If you already have a decentralized application (dApp) and are looking for a platform to help you scale efficiently, then you are at the right place because Polygon allows you to: 1. Easily migrate from Ethereum Virtual Machine (EVM) based chain: Polygon prides itself in being the ultimate Layer-2 scaling solution for Ethereum. You don't have to worry about the underlying architecture while moving or deploying your dApps to the Polygon PoS network as long as it is EVM-compatible. 2. Use Polygon PoS as a faster transaction layer: Deploying your dApp to the PoS mainnet allows you to leverage Polygon as a faster transaction layer for your dApp. Additionally, you can get your tokens mapped by us. You can join ourÂ technical discussions group on TelegramÂ to learn more. # matic-to-pol.md: Overview The technical upgrade from MATIC to POL marks a critical juncture for the Polygon networks, enhancing utility and aligning with the vision as an aggregated network of blockchains. POL will serve as a hyperproductive token: the native gas and staking token on Polygon PoS, as well as supporting the network's future expansion and security as an aggregated network. Steps to migrate to POL MATIC tokens on Ethereum !!! info "Stakers and delegators" MATIC stakers don't need to take any action to upgrade from MATIC to POL. If your MATIC tokens are on Ethereum, you can use Polygon Portal's migration interface to migrate your MATIC tokens to POL. The process is as follows: 1. Navigate to Polygon Portal's migration interface: https://portal.polygon.technology/pol-upgrade 2. Switch to Ethereum network in your wallet and connect to the Portal UI. 3. Approve the migration action by granting the upgrade contract permission to access your MATIC tokens. 4. Perform the migration action to receive POL in your wallet. MATIC tokens on Polygon PoS If your MATIC tokens are stored in your wallet on the Polygon PoS chain, you won't need to manually migrate them â€" they'll be automatically converted to POL at a 1:1 ratio. However, you'll need to update the native token symbol in your wallet's network settings. If the token symbol isn't updated, the wallet may continue to display MATIC as the token name instead of POL. Hereâ€™s how to do this in MetaMask. 1. Within your browser, open your MetaMask wallet in the expanded mode by selecting on the Expand view option from the options menu in the top-right corner.

!change-token-name-1{width=50%}

2. Select the options menu again from the wallet's expanded view, and then select Settings from the drop-down list. !change-token-name-2 3. Select the Networks tab from left sidebar to bring up the network settings. Switch to Polygon Mainnet if you're currently on another network. The list of configuration options on the right shows the Currency symbol which is currently set to MATIC. !change-token-name-3 4. Change the Currency symbol to POL, and select Save at the bottom. You can ignore the warning in yellow in this case.

!change-token-name-4{width=50%}

The process to change the token symbol may vary depending on the wallet you're using. Please refer to the docs specific to your wallet and follow the outlined steps accordingly. MATIC tokens on Polygon zkEVM If your MATIC tokens are on the zkEVM chain, use Polygon Portal to bridge your tokens to Ethereum, and then follow the steps described in the MATIC tokens on Ethereum section. Read more about POL 1. Detailed blog post on MATIC to POL migration 2. POL token reference doc # governance-fundamentals.md: The Polygon PoS chain is a decentralized network of validator nodes that participate in block generation and consensus, and non-validator nodes that perform functions such as maintaining the complete block history of the network, providing dApps with an interface to communicate with the chain, and so on. No single node controls the network, meaning consensus is required for upgrades. Changes to the network require a high level of coordination and ecosystem consensus to execute successfully; if the ecosystem disagrees over a change, it can result in the network splitting, a protocol-level change generally referred to as forking. !!! info "Hard forks vs. soft forks" A hard fork happens when the node software changes in such a way that the new version is no longer backward-compatible with earlier blocks. This is usually the result of a change in the consensus logic, meaning that blocks validated using the latest software will produce a different hash. A block number is selected, before which all nodes in the network should have upgraded to the new version; nodes running the old version will be disconnected from the canonical chain after the hard fork block. Should there be ${1/3}+1$ staked POL in disagreement with the fork, two canonical chains will temporarily form until the end of the current span. Afterwards, Bor will stop producing blocks, and the chain will halt until consensus is reached. In contrast, a soft fork is backward-compatible with the pre-fork blocks. This type of protocol change does not require nodes to upgrade before a deadline, therefore, multiple versions of the node software can be running at once and be able to validate transactions. The key ecosystem stakeholders involved in implementing a change are: - Users - Token holders - Validators - Infrastructure providers - Full nodes - Core developers The PoS chain uses an improvement proposal-based framework to meet these coordination requirements. This framework acts as a signaling mechanism for stakeholders, and is not binding. Ecosystem consensus The preliminary ecosystem consensus takes place through an off-chain process involving key stakeholders, including users and core developers. The framework set in place accommodates different perspectives put forward by the stakeholders, and provides a platform for constructive discussions and community cohesion. The framework is composed of three key components: 1. Polygon Improvement Proposals (â€œPIPsâ€): Outlined in PIP-1, PIPs are essentially instruments that enable the community to put forward protocol upgrades in the form of formal proposals that aim to improve the network. The framework borrows heavily from Ethereum and EIP-1, with many guiding principles originating within the IETF and the broader open-source community. 2. Polygon Protocol Governance Call (â€œPPGCâ€): Synchronous discussions where "rough consensus" is established, and the technical community makes protocol decisions. These calls generally decide which PIPs will be included in a particular upgrade, along with the roll-out schedule. 3. Polygon Community Forum: A space for long-form discussions ranging from high-level meta discussions to low-level technical details. Implementation The PoS network currently uses two clients simultaneously: - Heimdall: The consensus layer client - See GitHub - Bor: The execution layer client - See GitHub Currently, Bor and Heimdall are the majority clients for the PoS network. These clients serve as ecosystem focal points rather than control switches operated by core developers that can dictate decisions. Assuming that the change or upgrade agreed upon via community consensus requires a hard fork, the process that ensues generally looks like this: 1. The protocol decision is made on a PPGC, and implementation begins in the form of modifications to the relevant GitHub repositories. 2. Core developers create pull requests containing the changes, which can then be merged into the respective code base, and a new tag is created. 3. Core developers test new releases by deploying them on local devnets. If everything continues to function normally, the tag is marked as beta, which is essentially the pre-release state. 4. The modifications and upgrades are rolled out to the Amoy testnet, and left out to soak for at least one week. Currently, the Amoy Testing Committee reports on the stability of the release in the PPGC. 5. Finally, once confirmed that the upgrade doesn't break anything, it is scheduled to be released to mainnet on a PPGC. At this point, the tag version is marked as final. 6. Validators upgrade their nodes to the latest version after considering the changes. The upgrade is now made canonical via on-chain consensus of the validating stake, including that delegated by token holders. On-chain consensus The parameters that define on-chain consensus are inherited from Tendermint BFT, which requires at least ${2/3}rd$ of the total validating stake to be in favour of the upgrade. For the chain to remain stable once the change is made canonical by validators, non-validating full nodes must also be upgraded to the latest version. Key ecosystem stakeholders such as dApps, exchanges, and RPCs run full nodes, and are crucial in network operations as they propagate transactions and blocks. These nodes can either accept or reject blocks. This makes them enforcers of the network consensus rules and vital to the on-chain governance process. Should these nodes be incompatible with the changes, users and dApps would find that their transactions are invalid and not accepted by the network. On-chain governance module The Heimdall client also has an in-built governance module that can synchronously carry out consensus parameter changes across the network. 1. Proposals can be submitted to the on-chain module along with a deposit containing the proposed changes. 2. Each validator then tallies votes cast by validators. 3. When the defined voting parameters are met, each validator makes the upgrade with the proposal data. The current voting parameters (denominated in staked POL): - Quorum: 33.4% - Threshold: 50% - Veto: 33.4% A list of the changeable parameters is available here. # delegate.md: How to delegate This is a step-by-step guide to help you become a delegator in the Polygon network. The only prerequisite is to have your POL tokens and ETH on the Ethereum mainnet address. !!! info "Staking MATIC" Polygon network is transitioning from MATIC to POL, which will serve as the gas and staking token on Polygon PoS. Use the links below to learn more: - Migrate from MATIC to POL - POL token specs It is advisable to migrate your MATIC tokens to POL, but if you continue to delegate MATIC tokens, you'll receive the staking rewards in the form of POL tokens. Access the dashboard 1. In your wallet (e.g. MetaMask), choose the Ethereum mainnet.

!Figure: Choose ERTHh mainnet {width=50%}

2. Log in to Polygon Staking. 3. Once you log in, you will see overall statistics along with the list of validators.

!img

!!! note If you are a validator, use a different non-validating address to log in as delegator. Delegate to a validator 1. Select Become a Delegator, or scroll down to a specific validator and select Delegate.

!img

2. Select POL or MATIC from the drop-down list and enter the token amount to delegate. It is recommended to migrate your MATIC tokens to POL and delegate POL tokens. If you choose MATIC, you'll still receive your staking rewards in POL. Then, select Continue.

!img{width=50%}

3. Approve the delegate transaction from your wallet and select Delegate.

!img{width=50%}

After the delegation transaction completes, you will see the Delegation Completed message.

!img{width=50%}

View your delegations To view your delegations, select My Account. !img Withdraw rewards 1. Select My Account.

!img{width=70%}

2. Under your delegated validator, select Withdraw Rewards.

!img

This will withdraw the POL token rewards to your Ethereum address. Restake rewards 1. Select My Account.

!img{width=70%}

2. Under your delegated validator, click Restake Reward. !img This will restake the POL token rewards to the validator and increase your delegation stake. Unbond from a validator 1. Select My Account.

!img{width=70%}

2. Under your delegated validator, select Unbond. !img This will withdraw your rewards from the validator and your entire stake from the validator. Your withdrawn rewards will show up immediately in your Ethereum wallet. Your withdrawn stake funds will remain locked for 80 checkpoints. !img{width=50%} !!! note The fund locking for the unbonding period is in place to ensure there is no malicious behaviour on the network. Move stake from one node to another node Moving stake from one node to another node is a single transaction. There are no delays or unbonding periods during this event. 1. Select My Account and login to the staking dashboard. 2. Select

Move Stake under your delegated validator. 3. Select an external validator and select Stake here. !img 4. Provide the stake amount and select Move Stake.

!img{width=50%}

This will move the stake. The dashboard will update after 12 block confirmations. Common queries and solutions What is the staking dashboard URL? The staking dashboard URL is https://staking.polygon.technology/. What is the minimum stake amount? There is no minimum stake amount to delegate. However, you can always start with 1 POL token. How to stake tokens on Polygon? For staking, you would need to have funds on the Ethereum mainnet (more information here). Log into your wallet on the Ethereum network using the staking dashboard. Please watch this video for a graphical illustration of how this works:

Your browser does not support the video element.

Why does my transaction take so long? All staking transactions of Polygon PoS take place on Ethereum for security reasons. The time taken to complete a transaction depends on the gas fees that you have allowed and also the network congestion of Ethereum mainnet at that point in time. You can always use the Speed Up option to increase the gas fees so that your transaction can be completed soon. I've staked my POL tokens. How can I stake more? Navigate to the Your Delegations page and choose one of the stakes. Then click on Stake More. Please watch this video for a graphical illustration of how this works:

Your browser does not support the video element.

Why am I not able to stake? Check if you have funds on the Main Ethereum Network, to delegate your tokens. All staking happens on the Ethereum Network only. I am unable to view the staking tab. How do I access staking? You just need to access https://staking.polygon.technology/, where you will see the following landing page: !img How do I know which validator to select for better rewards? It depends on your understanding and research on which validator you would want to stake on. You can find the list of validators here : https://staking.polygon.technology/validators How to unbond? To unbond from a validator, navigate to My Account, where you'll find Your Delegations. There you will see an Unbond button for each of the validators. Click on the Unbond button for the validator that you want to unbond from. !img Please watch the video for a graphical illustration of how this works:

Your browser does not support the video element.

What is the unbonding period? The unbonding period on Polygon PoS is 80 checkpoints. Every checkpoint takes approximately 30 minutes. However, some checkpoints could be delayed due to congestion on Ethereum. This period applies to the originally delegated amount and re-delegated amounts. It does not apply to any rewards that were not re-delegated. How to restake rewards? Go to My Account to check Your Delegations. Clicking on Restake Reward will ask you for confirmation from your wallet account. Once you confirm the transaction in your wallet, the restake transaction is completed. Step 1

!img{width=50%}

Step 2

!img{width=50%}

Please watch the video for a graphical illustration of how this works:

Your browser does not support the video element.

I want to restake rewards but I am unable to. You'll need to have a minimum of 2 POL to restake rewards. How to withdraw rewards? You can claim your rewards by clicking on the My Account, all the delegators for a validator are displayed. Click on the Withdraw Reward button and the rewards will be transferred to your delegated account in wallet. Step 1

!img{width=50%}

Step 2

!img{width=50%}

Please watch the video for a graphical illustration of how this works:

Your browser does not support the video element.

I want to withdraw rewards but I am unable to. You'll need to have a minimum of 2 POL available to withdraw rewards. How to claim stake? Once the unbonding period is complete, the Claim Stake button will be enabled and you can then claim your staked tokens. The tokens will be transferred to your account. Step 1

!img{width=50%}

Step 2

!img

Step 3

!img{width=50%}

Please watch the video for a graphical illustration of how this works:

Your browser does not support the video element.

Which wallets are currently supported? We have recently upgraded the wallet support to WalletConnect v2.0. Now you can choose from a plethora of wallets, including Metamask, Coinbase, and others, on both desktop and mobile devices to log in.

!Figure: Supported wallet{width=50%}

Are hardware wallets supported? Yes, hardware wallets are supported. You can use the Connect Hardware Wallet option on MetaMask and connect your hardware wallet and then continue the delegation process. Why canâ€™t I stake directly from Binance? Staking through Binance is not yet supported. There will be an announcement if and when Binance starts supporting it. I have completed my delegation, where can I check details? Once you have completed your delegation, wait for 12 block confirmations on Ethereum (approx. 3-5 minutes), then on the dashboard, you can click on My Account. !Figure: My account Where can I check my rewards? On the dashboard, you can click on the My Account option on the left-hand side. !Figure: My account Do I need ETH to pay for gas fees? Yes. You should maintain at least 0.05-0.1 ETH balance for gas fees to be safe. Do I need to deposit POL tokens to the Polygon mainnet network for staking? No. All your funds need to be on the main Ethereum network. When I try to do the transaction my Confirm button is disabled. Please check if you have enough ETH for the gas fees. When do rewards get distributed? The rewards are distributed whenever a checkpoint is submitted. Currently, 71795 POL tokens are distributed proportionately on each successful checkpoint submission to each delegator based on their stake relative to the overall staking pool of all validators and delegators. Also, the percentage for the reward distributed to each delegator will vary with each checkpoint depending on the relative stake of the delegator, validator and the overall stake. Note that there is a 10% proposer bonus that accrues to the validator who submits the checkpoint, but over time, the effect of the extra bonus is nullified over multiple checkpoints by different validators. The checkpoint submission is done by one of the validators approximately every 30 minutes. This time is approximate and may vary based on validator consensus on the Polygon Heimdall layer. This may also vary based on Ethereum Network. Higher congestion in the network may result in delayed checkpoints. You can track checkpoints on the staking contract here Why do rewards keep getting decreased at every checkpoint? Actual rewards earned will depend on the actual total locked supply in the network at each checkpoint. This is expected to vary significantly as more POL tokens get locked in the staking contracts. Rewards will be higher, to begin with, and will keep decreasing as the locked supply % goes up. This change in locked supply is captured at every checkpoint, and rewards are calculated based on this. How can I claim my rewards? You can claim your rewards instantly by clicking on the Withdraw Reward button. This will transfer the rewards accumulated to your delegated account on Metamask. !Figure: Withdraw-reward Will I keep receiving rewards after I unbond? No. Once you unbond, you will stop receiving rewards. How many transactions does the delegation require? The delegation action takes place in 2 successive transactions. The first transaction to approve the request, and the second to deposit the approved token amount. !Figure: Delegate What does re-delegate rewards mean? Re-delegating your rewards simply means that you want to increase your stake by restaking the rewards you have accumulated. Can I stake to any validator? Yes, you can delegate to any validator except the ones which are offline. Can I move the stake to another validator? Yes, you just have to access Your Delegations, click on Move Stake, and then choose your new validator. Please watch the video for a graphical illustration of how this works:

Your browser does not support the video element.

Which browser is compatible with the Polygon earnings calculator? Chrome, Firefox, and Brave. My MetaMask is stuck at confirming after login, what do I do? Or nothing happens when I try to login? Check for the following: - If youâ€™re using Brave, please turn off the option for Use Crypto Wallets in the settings panel. - Check if you are logged into Metamask - Check if you are logged into MetaMask with Trezor/Ledger. You need to additionally turn on permission to call contracts on your Ledger device, if not enabled already. - Check your system timestamp. If the system time is not correct, you will need to correct it. How do I send funds from Binance or other exchanges to Polygon wallet? The Polygon Wallet Suite is a web application. First, you must withdraw your funds from Binance or any other exchange to your Ethereum address on Metamask. If you don't know how to use Metamask, google it a bit. There are plenty of videos and blogs to get started with it. When can I become a validator and how many tokens do I need for that? To become a validator in the PoS network, you'll need to hold and stake a minimum of 10,000 POL tokens, and go through an admissions process to ensure network security." If I have earned rewards while delegating, and if I add additional funds to the same validator node, what happens? If you have not re-delegated your rewards before delegating additional funds to the same validator node, your rewards will be withdrawn automatically. In case you don't want that to happen, re-delegate your rewards before delegating additional funds. I have delegated my tokens via MetaMask on the dashboard. Do I need to keep my system or device on? No. Once your delegation transactions are confirmed, and you can see your tokens reflected in the Total Stake and New Reward sections, then you are done. There is no need to keep your system or device on. I have unbonded, how long will it take to unbond? The unbonding period is currently set to 80 checkpoints. Every checkpoint takes approximately 30 minutes. However, some checkpoints could be delayed upto 1 hour due to congestion on Ethereum. I have unbonded, and I now see the Claim Stake button, but it is disabled. The Claim Stake button will only be enabled when your unbonding period is complete. The unbonding period is currently set at 80 checkpoints. When will the Claim Stake button be enabled? Under the Claim Stake button you should see a note on how many checkpoints are pending before the Claim Stake button is enabled. Every checkpoint takes approximately 30 minutes. However, some checkpoints could be delayed upto 1 hour due to congestion on Ethereum. !Figure: Unbound {width=50%} Have the foundations nodes been turned off? Yes, the foundation nodes had been turned off. Will there be any foundation nodes in the future? No, there won't be any Foundation nodes in the future. How many transactions do I need to pay for gas when I perform a 'move stake' action? The 'move stake' operation is performed in a single transaction. Since all transactions are executed on the Ethereum blockchain, you will need to spend some ETH for gas fees to complete the transaction. # erigon-archive-node.md: System requirements - CPU: 16-core, 64-bit architecture - RAM: 64GB - Storage - Basically io1 or above with at least 20k+ iops and RAID-0 based disk structure - Mainnet archive node: 15TB - Amoy testnet archive node: 1TB - SSD or NVMe. Bear in mind that SSD performance deteriorates when close to capacity. - Golang: >= v1.20 - GCC: >= v10 !!! tip "HDD not recommended" On HDDs, Erigon will always remain N blocks behind the chain tip, but will not fall further behind. Install Erigon client Run the following commands to install Erigon: bash git clone --recurse-submodules -j8 https://github.com/ledgerwatch/erigon cd erigon git checkout v2.57.3 make erigon This should create the binary at ./build/bin/erigon Start Erigon client If you're deploying to mainnet, run the following command: bash erigon --chain=bor-mainnet --db.size.limit=12TB --db.pagesize=16KB remaining flags follow When connecting to Amoy testnet, use the following command to start your Erigon client: bash erigon --chain=amoy Configure Erigon client If you want to store Erigon files in a non-default location, use -datadir to specify a new location: bash erigon --chain=amoy --datadir= If you are not using local heimdall, use -bor.heimdall=. By default, it will try to connect to localhost:1317. bash erigon --chain=amoy --bor.heimdall= --datadir= Node RPC - If you want to connect to PoS Amoy Testnet, use: https://heimdall-api-amoy.polygon.technology - For PoS mainnet, use: https://heimdall-api.polygon.technology !!! tip Remote heimdall is better suited for testing, and is not recommended for production use. Tips for faster sync - Use the machine with high IOPS and RAM for the faster initial sync - Memory optimized nodes are recommended for faster sync. For example, AWS EC2 r5 or r6 series instances. Reporting issues In case you encounter any issues and are looking for support, please get in touch with the Erigon team. More details available in the Erigon GitHub README. # prerequisites.md: Node system requirements Please note that all system requirements listed below are presented in a Minimum/Recommended format. Mainnet specs | Node type | RAM | CPU | Storage | Network bandwidth | | -------------------- | ---------- | -------------- | -- | ---------------- | | Full Node/Sentry Node | 32 GB/64 GB | 8 core/16 core | 4 TB/6 TB | 1 Gbit/s | | Validator Node | 32 GB/64 GB | 8 core/16 core | 4 TB/6 TB | 1 Gbit/s | | Archive Node (Erigon) | 64 GB | 16 core | 16 TB(io1Â or above with at least 20k+ iops and RAID-0 based disk structure) | 1 Gbit/s | Testnet (Amoy) specs | Node type | RAM | CPU | Storage | Network bandwidth | | -------------------- | ---------- | -------------- | -- | ---------------- | | Full Node/Sentry Node | 8 GB/16 GB | 8 core/16 core | 1 TB/2 TB | 1 Gbit/s | | Validator Node | 8 GB/16 GB | 8 core/16 core | 1 TB/2 TB | 1 Gbit/s | | Archive Node (Erigon) | 16 GB | 16 core | 1 TB/2 TB (io1Â or above with at least 20k+ iops and RAID-0 based disk structure) | 1 Gbit/s | Downloading the snapshot It is recommended that you keep your snapshots handy before setting up the node. Link to the snapshot documentation here. Open necessary ports Sentry/full nodes | Port | Description | | :--------------------: | -- | | 26656 | Heimdall service connects your node to another nodeâ€™s Heimdall service using this port. | | 30303 | Bor service connects your node to another nodeâ€™s Bor service using this port. | | 22 | For the validator to be able to SSH from wherever they are. | | 26660 | Prometheus port for Tendermint/Heimdall. Not required to be opened to the public. Only allow for the monitoring systems (Prometheus/Datadog). | | 7071 | Metric port for Bor. Only needs to be opened for the Monitoring system. | | 8545,Â 8546, Â 1317 | Can be opened for Bor HTTP RPC, Bor WS RPC, and Heimdall API respectively; but only if really necessary. | Validator nodes | Port | Description | | :-----: | -- | | 22 | Ope this to the public is not a good idea as the default SSH port 22 is prone to attacks. It is better to secure it by allowing it only in a closed network (VPN). | | 30303 | To be opened to only Sentry to which the validator is connected for Bor P2P discovery. | | 26656 | To be opened to only Sentry to which the validator is connected for Heimdall/Tendermint P2P discovery. | | 26660 | Prometheus port for Tendermint/Heimdall. Not required to be opened to the public. Only allow for the monitoring systems (Prometheus/Datadog). | | 7071 | Metric port for Bor. Only needs to be opened for the monitoring system. | Install RabbitMQ !!! info "Only for validator nodes" This step is only relevant for validator nodes. Before setting up your validator node, itâ€™s advisable to install the RabbitMQ service. You can use the following commands to set up RabbitMQ (if itâ€™s not already installed): bash sudo apt-get update sudo apt install build-essential sudo apt install erlang wget https://github.com/rabbitmq/rabbitmq-server/releases/download/v3.10.8/rabbitmq-server3.10.8-1all.deb sudo dpkg -i rabbitmq-server3.10.8-1all.deb Connect to Ethereum RPC endpoint !!! info "Only for validator nodes" This step is only relevant for validator nodes. Validator nodes need to connect to an Ethereum RPC endpoint. You may use your own Ethereum node, or utilize external infrastructure providers. Mandatory checklist for validators Please follow the below checklist in order to set up your validator node using binaries, Ansible, or packages. | Checklist | Binaries | Ansible | Packages | | ------------------ | -- | -- | -- | | Machines required | 2 Machines -Â sentryÂ &Â validator | 3 Machines -Â local machine,Â sentryÂ andÂ validator | 2 Machines -Â sentryÂ &Â validator | | Install Go packages | Yes | No | No | | Install Python | No | Yes (only on the local machine where the Ansible Playbook runs) | No | | Install Ansible | No | Yes (only on one machine) | No | | Install Bash | No | No | Yes | | Run Build Essential | Yes | No | No | | Node setup | Using binaries | Using Ansible | Using packages | # snapshots.md: When setting up a new sentry, validator, or full node server, it is recommended that you use snapshots for faster syncing without having to sync over the network. Using snapshots will save you several days for both Heimdall and Bor. Community snapshots Polygon PoS has transitioned to a community-driven model for snapshots. Active community members now contribute to provide snapshots. Some of these members include: | Name | Available snapshots | Note | | -- | -------------------- | -- | | Stakecraft | Mainnet, Amoy, Erigon | Support for Erigon archive snapshot | | PublicNode (by Allnodes) | Mainnet, Amoy | Support for PBSS + PebbleDB enabled snapshot | | Stakepool | Mainnet, Amoy | - | | Vaultstaking | Mainnet | - | | Girnaar Nodes | Amoy | - | > \The PBSS + PebbleDB snapshot provided by PublicNode is currently in the beta phase. !!! info "Snapshot aggregator" Visit All4nodes.io for a comprehensive list of community snapshots. Downloading and using client snapshots To begin, ensure that your node environment meets the prerequisites outlined here. The majority of snapshot providers have also outlined the steps that need to be followed to download and use their respective client snapshots. Navigate to All4nodes to view the snapshot source. In case the steps are unavailable or the procedure is unclear, the following tips will come in handy: - You can use the wget command to download and extract the .tar snapshot files. For example: bash wget -O - snapshoturlhere | tar -xvf -C /target/directory - Configure your client's datadir setting to match the directory where you downloaded and extracted the snapshot data. This ensures the systemd services can correctly register the snapshot data when the client is spun up. - To maintain your client's default configuration settings, consider using symbolic links (symlinks). Example Let's say you have mounted your block device at /snapshots and have downloaded and extracted the chain data into the heimdallextract directory for Heimdall, and into the borextract directory for Bor. Use the following commands to register the extracted data for Heimdall and Bor systemd services: bash remove any existing datadirs for Heimdall and Bor rm -rf /var/lib/heimdall/data rm -rf /var/lib/bor/chaindata rename and setup symlinks to match default client datadir configs mv /snapshots/heimdallextract /snapshots/data mv /snapshots/borextract /snapshots/chaindata sudo ln -s /snapshots/data /var/lib/heimdall sudo ln -s /snapshots/chaindata /var/lib/bor bring up clients with all snapshot data properly registered sudo service heimdalld start wait for Heimdall to fully sync then start Bor sudo service bor start !!! tip "Appropriate user permissions" Ensure that the Bor and Heimdall user files have appropriate permissions to access the datadir. To set correct permissions for Bor, execute sudo chown -R bor:nogroup /var/lib/heimdall/data. Similarly, for Heimdall, run sudo chown -R heimdall:nogroup /var/lib/bor/data/bor Recommended disk size guidance Polygon Amoy testnet | Metric | Calculation Breakdown | Value | | --------------------------------- | ---------------------------------- | ------- | | approx. compressed total | 250 GB (Bor) + 35 GB (Heimdall) | 285 GB | | approx. data growth daily | 10 GB (Bor) + 0.5 GB (Heimdall) | 10.5 GB | | approx. total extracted size | 350 GB (Bor) + 50 GB (Heimdall) | 400 GB | | suggested disk size (2.5x buffer) | 400 GB 2.5 (natural chain growth) | 1 TB | Polygon mainnet | Metric | Calculation Breakdown | Value | | --------------------------------- | ---------------------------------- | ------- | | approx. compressed total | 3000 GB (Bor) + 500 GB (Heimdall) | 3500 GB | | approx. data growth daily | 100 GB (Bor) + 5 GB (Heimdall) | 105 GB | | approx. total extracted size | 4 TB (Bor) + 500 GB (Heimdall) | 4.5 TB | | suggested disk size (2.5x buffer) | 4 TB 2 (natural chain growth) | 8 TB | Polygon Amoy Erigon archive | Metric | Calculation Breakdown | Value | | --------------------------------- | ----------------------------------- | ------ | | approx. compressed total | 210 GB (Erigon) + 35 GB (Heimdall) | 245 GB | | approx. data growth daily | 4.5 GB (Erigon) + 0.5 GB (Heimdall) | 5 GB | | approx. total extracted size | 875 GB (Erigon) + 50 GB (Heimdall) | 925 GB | | suggested disk size (2.5x buffer) | 925 GB 2.5 (natural chain growth) | 2.5 TB | Recommended disk type and IOPS guidance - Disk IOPS will affect the speed of downloading/extracting snapshots, getting in sync, and performing LevelDB compaction. - To minimize disk latency, direct-attached storage is ideal. - In AWS, when using gp3 disk types, we recommend provisioning IOPS of 16,000 and throughput of 1,000. This minimizes costs while providing significant performance benefits. io2 EBS volumes with matching IOPS and throughput values offer similar performance. - For GCP, we recommend using performance (SSD) persistent disks (pd-ssd) or extreme persistent disks (pd-extreme) with similar IOPS and throughput values as mentioned above. # ethereum-to-matic.md: !!! warning "Work in progress!" This doc is currently undergoing revision, and the instructions provided may not be up to date. Stay tuned for updates! The mechanism to natively read Ethereum data from Polygon EVM chain is that of â€˜State Syncâ€™. In other words, this mechanism enables transfer of arbitrary data from Ethereum chain to Polygon chain. The procedure that makes it possible is: Validators on the Heimdall layer are listening for a particular event â€" StateSynced from a sender contract, as soon as the event is picked, the data that was passed in the event is written on the receiver contract. Read more here. The sender and receiver contracts are required to be mapped on Ethereum â€" StateSender.sol needs to be aware of each sender and receiver. !!! tip "Custom tokens" Looking to bridge your custom token to Polygon PoS using the official bridge? Check out the guide on how to submit a request to get your token mapped. --- In the following walkthrough, we'll be deploying a sender contract on Sepolia (Ethereum testnet) and a receiver contract on Amoy (Polygon testnet). Then, we'll be sending data from the sender and reading data on the receiver via web3 calls in a node script. 1. Deploy sender contract The sole purpose of the sender contract is to be able to call syncState function on the StateSender contract â€" which is the state syncer contract on Polygon PoS that emits the StateSynced event that Heimdall listens for. Deployed at: 0x49E307Fa5a58ff1834E0F8a60eB2a9609E6A5F50 on Sepolia 0x28e4F3a7f651294B9564800b2D01f35189A5bFbE on Ethereum Mainnet To be able to call this function, let's first include it's interface in our contract: jsx title="Sender.sol" pragma solidity ^0.5.11; contract IStateSender { function syncState(address receiver, bytes calldata data) external; function register(address sender, address receiver) public; } ... Next, let's write our custom function that takes in the data we'd like to pass on to Polygon and calls syncState. jsx function sendState(bytes calldata data) external { states = states + 1 ; IStateSender(stateSenderContract).syncState(receiver, data); } In the above function, stateSenderContract is the address of the StateSender on the network you'll be deploying the sender on. (eg., we'll be using 0x49E307Fa5a58ff1834E0F8a60eB2a9609E6A5F50 for Sepolia), and the receiver is the contract that receives the data we send from here. It is recommended to use constructors to pass in variables, but for the purpose of this demo, we'll simply hardcode these two addresses: Following is what our Sender.sol looks like: jsx title="Sender.sol" pragma solidity ^0.5.11; contract IStateSender { function syncState(address receiver, bytes calldata data) external; function register(address sender, address receiver) public; } contract sender { address public stateSenderContract = 0xEAa852323826C71cd7920C3b4c007184234c3945; address public receiver = 0x83bB46B64b311c89bEF813A534291e155459579e; uint public states = 0; function sendState(bytes calldata data) external { states = states + 1 ; IStateSender(stateSenderContract).syncState(receiver, data); } } We're using a simple states counter to keep track of the number of states sent via the sender contract. Use Remix to deploy the contract and keep a note of the address and ABI. 2. Deploy receiver contract The receiver contract is the one that is invoked by a validator when the StateSynced event is emitted. The validator invokes the function onStateReceiveon the receiver contract to submit the data. To implement it, we first import StateReceiver interface and write down our custom logic â€" to interpret the transferred data inside onStateReceive. The following is what our Receiver.sol looks like: jsx title="Receiver.sol" pragma solidity ^0.5.11; // IStateReceiver represents interface to receive state interface IStateReceiver { function onStateReceive(uint256 stateId, bytes calldata data) external; } contract receiver { uint public lastStateId; bytes public lastChildData; function onStateReceive(uint256 stateId, bytes calldata data) external { lastStateId = stateId; lastChildData = data; } } The function simply assigns the last received state ID and data to variables. StateId is a simple unique reference to the transferred state (a simple counter). Deploy your Receiver.sol to Amoy testnet and keep a note of the address and ABI. 3. Getting your sender and receiver contracts mapped You can either use the already deployed addresses (mentioned above) for the sender and receiver, or deploy your custom contracts and request a mapping using the Google form here. 4. Sending and receiving data Now that we have our contracts in place and mapping done, we'll be writing a simple node script to send arbitrary hex bytes, receive them on Polygon and interpret the data! Setup your script We'll first initialize our web3 objects, wallet to make the transactions and contracts. jsx title="test.js" const Web3 = require('web3') const Network = require("@maticnetwork/meta/network") const network = new Network ('testnet', 'amoy') const main = new Web3(network.Main.RPC) const matic = new Web3 (network.Matic.RPC) let privateKey = 0x... // add or import your private key matic.eth.accounts.wallet.add(privateKey) main.eth.accounts.wallet.add(privateKey) let receiverAddress = let receiverABI = // insert or import ABI let senderAddress = let senderABI = // insert of import the ABI let sender = new main.eth.Contract(JSON.parse(senderABI), senderAddress) let receiver = new main.eth.Contract(JSON.parse(receiverABI), receiverAddress) We're using @maticnetwork/meta package for the RPCs, the package isn't a requirement to run the script. matic and main objects refer to the web3 object initialized with Polygon Amoy and Sepolia network's respective RPC URLs. sender and receiver objects refer to the contract objects of Sender.sol and Receiver.sol that we deployed in Step 1 and 2. Sending data Next, let's setup our functions to create bytestring of the data and send it via the sender contract: jsx // data to sync function getData(string) { let data = matic.utils.asciiToHex(string); return data } // send data via sender async function sendData (data) { let r = await sender.methods .sendState (getData(data)) .send({ from: main.eth.accounts.wallet[0].address, gas: 8000000 }) console.log('sent data from root, ', r.transactionHash) } Calling getData will convert an ASCII string (eg., Hello World !) to a string of bytes (eg., 0x48656c6c6f20576f726c642021); while the function sendDatatakes in data (an ascii string), calls getData and passes on the bytestring to sender contract Receiving data Next, we'll be checking for received data on Receiver.sol. It should take 7-8 minutes for the state sync to execute. Add the following functions to check the number of sent states from sender, and the last received state on the receiver contract. jsx // check states variable on sender async function checkSender () { let r = await sender.methods .states() .call() console.log('number of states sent from sender: ', r) } // check last received data on receiver async function checkReceiver () { let r = await receiver.methods .lastStateId() .call() let s = await receiver.methods .lastChildData() .call() console.log('last state id: ', r, 'and last data: ', s) console.log('interpreted data: ', getString(s)) } The function checkReceiver simply calls the variables we defined in the contract â€" which would be set as soon as the Validator calls onStateReceive on the contract. The getString function simply interprets the bytestring (converts it back to ASCII) jsx function getString(data) { let string = matic.utils.hexToAscii(data); return string } Finally, we'll write up a method to execute our functions: jsx async function test() { await sendData ('Sending a state sync! :) ') await checkSender () await checkReceiver () } Putting it all together! This is how our test script looks like: jsx title="test.js" const Web3 = require('web3') const Network = require("@maticnetwork/meta/network") const network = new Network ('testnet', 'amoy') const main = new Web3(network.Main.RPC) const matic = new Web3 (network.Matic.RPC) let privateKey = 0x... matic.eth.accounts.wallet.add(privateKey) main.eth.accounts.wallet.add(privateKey) let receiverAddress = let receiverABI = let senderAddress = let senderABI = let sender = new main.eth.Contract(JSON.parse(senderABI), senderAddress) let receiver = new matic.eth.Contract(JSON.parse(receiverABI), receiverAddress) // data to sync function getData(string) { let data = matic.utils.asciiToHex(string); return data } function getString(data) { let string = matic.utils.hexToAscii(data); return string } // console.log(getData('Sending a state sync! :) ')) async function sendData (data) { let r = await sender.methods .sendState (getData(data)) .send({ from: main.eth.accounts.wallet[0].address, gas: 8000000 }) console.log('sent data from root, ', r.transactionHash) } async function checkSender () { let r = await sender.methods .states() .call() console.log('number of states sent from sender: ', r) } async function checkReceiver () { let r = await receiver.methods .lastStateId() .call() let s = await receiver.methods .lastChildData() .call() console.log('last state id: ', r, 'and last data: ', s) console.log('interpreted data: ', getString(s)) } async function test() { await sendData ('Hello World !') await checkSender () // add a timeout here to allow time gap for the state to sync await checkReceiver () } test() Let's run the script Successful execution of the above script provide an output as: bash $ node test > sent data from root 0x4f64ae4ab4d2b2d2dc82cdd9ddae73af026e5a9c46c086b13bd75e38009e5204 number of states sent from sender: 1 last state id: 453 and last data: 0x48656c6c6f20576f726c642021 interpreted data: Hello World ! # matic-to-ethereum.md: !!! warning "Work in progress!" This doc is currently undergoing revision, and the instructions provided may not be up to date. Stay tuned for updates! The mechanism for transferring data from Polygon PoS to Ethereum differs from the process of transferring data from Ethereum to Polygon PoS. Validators create checkpoint transactions on the Ethereum chain to facilitate this transfer. These checkpoints serve as periodic summaries of the PoS chainâ€™s state, ensuring data integrity and consistency when moving data back to Ethereum. The flow of this process is briefly described below. 1. A transaction is created on Polygon PoS. It is crucial to emit an event and ensure that the event logs include the data intended for transfer to Ethereum. This process is essential for tracking and verifying the data transfer, as the event logs serve as a reliable record that can be referenced on the Ethereum network. 2. Within a time frame of approximately 10 to 30 minutes, this transaction is check-pointed on the Ethereum chain by the validators. 3. Once checkpointing is complete, the hash of the transaction created on PoS can be submitted as a proof on the RootChainManager contract on the Ethereum chain. This contract validates the transaction, verifies whether this transaction is included in the checkpoint, and finally decodes the event logs from this transaction. 4. The decoded event log data can then be used to perform changes on the root contract deployed on the Ethereum chain. 5. We use a predicate contract which is a special type of contract that can be only triggered by the RootChainManager contract to ensure the state update on Ethereum is secure. This architecture ensures that the state changes on Ethereum happens only when the transaction on Polygon is check pointed and verified on the Ethereum chain by the RootChainManager contract. Overview - A transaction is executed on the child contract deployed on the Polygon chain. - An event is also emitted in this transaction. The parameters of this event includes the data which has to be transferred from Polygon PoS to Ethereum. - The validators on the PoS network pick up this transaction in a specific interval of time(probably 10-30 mins), validate them and add them to the checkpoint on Ethereum. - A checkpoint transaction is created on the RootChain contract and the checkpoint inclusion can be checked using this script - Once the checkpoint addition is completed, the matic.js library can be used to call the exit function of the RootChainManager contract. Here's an example. - Running the script, verifies the inclusion of the Polygon transaction hash on Ethereum chain, and then in turn calls the exitToken function of the predicate contract. !!! tip "Securing state changes" The important thing to note is that the verification of the transaction hash from Polygon PoS and triggering the predicate contract happens in a single transaction, thus ensuring security of any state change in the root contract. Implementation This is a simple demonstration of how data can be transferred from Polygon PoS to Ethereum. This tutorial shows an example of transferring a uint256 value across the chain. But you can transfer any type of data. However, it is necessary to encode the data in bytes and then emit it from the child contract, which can finally be decoded by the root contract. First, create the root chain and child chain contract. Ensure that the function that does the state change also emits an event. This event must include the data to be transferred as one of its parameters. A sample format of how the child and root contract must look like is given below. This is a very simple contract that has a data variable whose value is set by using a setData function. Calling the setData function emits the Data event. Rest of the things in the contract will be explained in the upcoming sections of this tutorial. jsx title="Child contract" contract Child { event Data(address indexed from, bytes bytesData); uint256 public data; function setData(bytes memory bytesdata) public { data = abi.decode(bytesdata,(uint256)); emit Data(msg.sender,bytesdata); } } Pass this 0x1470E07a6dD1D11eAE439Acaa6971C941C9EF48f as the value for predicate in the root contract constructor. jsx title="Root contract" contract Root { address public predicate; constructor(address predicate) public{ predicate=predicate; } modifier onlyPredicate() { require(msg.sender == predicate); ; } uint256 public data; function setData(bytes memory bytesdata) public onlyPredicate{ data = abi.decode(bytesdata,(uint256)); } } Once the child and root contract is deployed on the Polygon and Ethereum chain respectively, these contracts have to be mapped using the PoS bridge. This mapping ensures that a connection is maintained between these two contracts across the chains. For doing this mapping,the Polygon team can be reached on Discord. One important thing to note is that in the root contract, there is a onlyPredicate modifier. It is recommended to use this modifier always because it ensures that only the predicate contract makes the state change on the root contract. The predicate contract is a special contract that triggers the root contract only when the transaction that happened on the Polygon PoS chain is verified by the RootChainManager on Ethereum chain. This ensures secure change of state on the root contract. For testing the above implementation, we can create a transaction on the Polygon chain by calling the setData function of the child contract. We need to wait at this point for the checkpoint to be completed. The checkpoint inclusion can be checked using this script. Once checkpoint is completed, call the exit function of the RootChainManager using matic.js SDK. jsx const txHash = "0xc094de3b7abd29f23a23549d9484e9c6bddb2542e2cc0aa605221cb55548951c"; const logEventSignature = "0x93f3e547dcb3ce9c356bb293f12e44f70fc24105d675b782bd639333aab70df7"; const execute = async () => { try { const tx = await maticPOSClient.posRootChainManager.exit(txHash, logEventSignature); console.log(tx.transactionHash); // eslint-disable-line } catch (e) { console.error(e); // eslint-disable-line } }; As shown in the above screenshot, the txHash is the transaction hash of the transaction that happened on the child contract deployed on Polygon chain. The logEventSignature is the keccack-256 hash of the Data event. This is the same hash that we have included in the predicate contract. All the contract code used for this tutorial and the exit script can be found here Once the exit script is completed, the root contract on Ethereum chain can be queried to verify if the value of the variable data that was set in child contract has also been reflected in the data variable of the root contract. # portal-ui.md: Polygon brings you a trustless two-way transaction channel between Polygon PoS and Ethereum by introducing the cross-chain bridge. The official bridge allows you to transfer tokens between Ethereum and Polygon PoS without incurring third-party risks and market liquidity limitations. The official bridge is available on both the PoS Amoy testnet as well as mainnet, and provides a near-instant, low-cost, and flexible bridging option for both users and dApp developers. You can access it and bridge assets from Ethereum over to Polygon PoS using Polygon Portal. !!! tip "Polygon Portal" To learn more about the features that Polygon Portal offers, and a series of step-by-step instructions that help you with using the platform, check out the Portal guide. There is no change to the circulating supply of your token when it crosses the bridge. This is what goes on in the background when you bridge your tokens over to Polygon PoS from Ethereum: - When depositing, tokens that leave the Ethereum network are locked and the same number of tokens are minted on Polygon PoS as a pegged token (1:1). - When withdrawing tokens back to the Ethereum network, tokens are burned on Polygon PoS and unlocked on Ethereum during the process. Additional resources - Introduction to Blockchain Bridges - What are Cross-Chain Bridges? # submit-mapping-request.md: Token mapping is important in order to enable the transfer mechanism for the said token between Ethereum and Polygon PoS. !!! info Thinking about bridging a popular token? Refer to the reference guide on Polygon Portal. Steps to submit a mapping request 1. To submit a request for mapping your token on Polygon PoS, start by navigating to the Google form available here. 2. Choose a Network option depending upon the chain on which you're looking to map your token. This would be Sepolia <> Polygon Amoy for testnet, and Ethereum <> Polygon PoS for mainnet.

!form-1{width=50%}

3. Next, input the contract address for the token contract that you've deployed on Sepolia/Ethereum mainnet in the Root Contract Address (L1) field.

!form-2{width=50%}

4. Choose the correct Token Type for your token. i.e., ERC-20 for a standard token, ERC-721 for an NFT, or ERC-1155 for a multi token.

!form-3{width=20%}

5. Finally, select Submit to send in your request. The Polygon team will review your mapping request, and get back to you with a response. This generally takes up to 7 days. !!! info "Token list" - Check out the list of supported tokens available in JSON format by following this URL: https://api-polygon-tokens.polygon.technology/tokenlists/polygonTokens.tokenlist.json - Once approved, your token will be added to the above list! # state-transfer.md: !!! warning "Work in progress!" This doc is currently undergoing revision, and the instructions provided may not be up to date. Stay tuned for updates! Polygon validators continuously monitor a contract on Ethereum chain called StateSender. Each time a registered contract on Ethereum chain calls this contract, it emits an event. Using this event Polygon validators relay the data to another contract on Polygon chain. This state sync mechanism is used to send data from Ethereum to Polygon. Additionally, Polygon validators send the transaction hash, namely checkpoint, of each transaction on the PoS chain to Ethereum on a regular basis.

You can use this to validate any transaction that took place on Polygon. Once a transaction has been verified to have occurred on the PoS chain, the corresponding action can then be executed on Ethereum. These two mechanisms can be used together to enable two-way data (state) transfer between Ethereum and Polygon. To abstract out all these interactions, you can directly inherit our FxBaseRootTunnel (on Ethereum) and FxBaseChildTunnel (on Polygon) contracts. Root tunnel contract Use the FxBaseRootTunnel contract from here. This contract gives access to the following functions: - function processMessageFromChild(bytes memory data): This is a virtual function that needs to be implemented in the contract which inherits it to handle data being sent from ChildTunnel. - sendMessageToChild(bytes memory message): This function can be called internally with any bytes data as a message. This data will be sent as it is to the child tunnel. - receiveMessage(bytes memory inputData): This function needs to be called to receive the message emitted by ChildTunnel. The proof of transaction needs to be provided as calldata. An example script to generate proof using the matic.js SDK is included below. Child tunnel contract Use the FxBaseChildTunnel contract from here. This contract gives access to following functions: - function processMessageFromRoot(uint256 stateId, address sender, bytes memory data): This is a virtual function that needs to implement the logic to handle messages sent from the RootTunnel. - function sendMessageToRoot(bytes memory message): This function can be called internally to send any bytes message to the root tunnel. Prerequisites You need to inherit FxBaseRootTunnel contract in your root contract on Ethereum. As an example, you can follow this contract . Similarly, inherit FxBaseChildTunnel contract in your child on Polygon. Follow this contract as an example. - While deploying your root contract on - Sepolia testnet, pass the address of checkpointManager as 0xbd07D7E1E93c8d4b2a261327F3C28a8EA7167209 and fxRoot as 0x0E13EBEdDb8cf9f5987512d5E081FdC2F5b0991e. - Ethereum mainnet, checkpointManager is 0x86e4dc95c7fbdbf52e33d563bbdb00823894c287 and fxRoot is 0xfe5e5D361b2ad62c541bAb87C45a0B9B018389a2. - For deploying the child contract on - Amoy testnet, pass 0xE5930336866d0388f0f745A2d9207C7781047C0f as fxChild in constructor. - Polygon mainnet, fxChild will be 0x8397259c983751DAf40400790063935a11afa28a. - Call setFxChildTunnel on deployed root tunnel with the address of child tunnel. Example: 0x97482d379e397329ac1ee2a34eeb9aceb06bd4a91ec17c7d7d3da4a1e96c165c - Call setFxRootTunnel on deployed child tunnel with address of root tunnel. Example: 0xae30445301bd7c902bf373fb890faf5658bd3a9437131c9408d5ecbc41af3fc0 State tunnel sample contracts - Contracts: Fx-Portal Github Repository - Sepolia: 0x1707157b9221204869ED67705e42fB65e026586c - Amoy: 0xf5D2463d0176462d797Afcd57eC477b7B0CcBE70 State transfer from Ethereum to Polygon - You need to call sendMessageToChild() internally in your root contract and pass the data as an argument to be sent to Polygon. Example: 0x00a1aa71593fec825b4b1ce1081b5a9848612fb21f9e56def2914b483f5f34f5 - In your child contract, implement processMessageFromRoot() virtual function in FxBaseChildTunnel to retrieve data from Ethereum. The data will be received automatically from the state receiver when the state is synced. State transfer from Polygon to Ethereum 1. Call sendMessageToRoot() internally in your child contract with data as a parameter to be sent to Ethereum. Note down the transaction hash as it will be used to generate the proof after the transaction has been included as a checkpoint. 2. Proof Generation to complete the exit on root chain: Generate the proof using the tx hash and MESSAGESENTEVENTSIG. To generate the proof, you can either use the proof generation API hosted by Polygon, or you can also spin up your own proof generation API by following the instructions here. The proof generation endpoint hosted by Polygon is available here: - Mainnet - Testnet Here, - burnTxHash is the transaction hash of the sendMessageToRoot() transaction you initiated on Polygon. - eventSignature is the event signature of the event emitted by the sendMessageToRoot() function. The event signature for the MESSAGESENTEVENTSIG is 0x8c5261668696ce22758910d05bab8f186d6eb247ceac2af2e82c7dc17669b036. Here's an example of how to use the proof generation API. 1. Implement processMessageFromChild() in your root contract. 2. Use the generated proof as an input to receiveMessage() to retrieve data sent from child tunnel into your contract. # full-node-ansible.md: An Ansible playbook can be used to configure and manage a full node. Prerequisites - Install Ansible on your local machine with Python3.x. The setup doesn't run on Python 2.x. - To install Ansible with Python 3.x, you can use pip. If you do not have pip on your machine, follow the steps outlined here. Run pip3 install ansible to install Ansible. - Check the Polygon PoS Ansible repository for requirements. - You also need to ensure that Go is not installed in your environment. You will run into issues if you attempt to set up your full node through Ansible with Go installed as Ansible requires specific packages of Go. - You will also need to make sure that your VM / Machine does not have any previous setups for Polygon Validator or Heimdall or Bor. You will need to delete them as your setup will run into issues. Full node setup - Ensure you have access to the remote machine or VM on which the full node is being set up. > Refer to https://github.com/maticnetwork/node-ansible#setup for more details. - Clone the https://github.com/maticnetwork/node-ansible repository. - Navigate into the node-ansible folder: cd node-ansible - Edit the inventory.yml file and insert your IP(s) in the sentry->hosts section. > Refer to https://github.com/maticnetwork/node-ansible#inventory for more details. - Check if the remote machine is reachable by running: ansible sentry -m ping - To test if the correct machine is configured, run the following command: bash Mainnet: ansible-playbook playbooks/network.yml --extra-var="borversion=v1.0.0 heimdallversion=v1.0.3 network=mainnet nodetype=sentry" --list-hosts Testnet: ansible-playbook playbooks/network.yml --extra-var="borversion=v1.1.0 heimdallversion=v1.0.3 network=amoy nodetype=sentry" --list-hosts !Figure: Full node testnet - Next, set up the full node with this command: bash Mainnet: ansible-playbook playbooks/network.yml --extra-var="borversion=v1.1.0 heimdallversion=v1.0.3 network=mainnet nodetype=sentry" Testnet: ansible-playbook playbooks/network.yml --extra-var="borversion=v1.0.0 heimdallversion=v1.0.3 network=amoy nodetype=sentry" - In case you run into any issues, delete and clean the whole setup using: bash ansible-playbook playbooks/clean.yml - Once you initiate the Ansible playbook, log in to the remote machine. - Please ensure that the value of seeds and bootnodes mentioned below is the same value as mentioned in Heimdall and Bor config.toml files. If not, change the values accordingly. !!! tip "Amoy testnet seeds" The Heimdall and Bor seeds don't need to be configured manually for Amoy testnet since they've already been included at genesis. - Heimdall seed nodes: bash moniker= Mainnet: seeds="1500161dd491b67fb1ac81868952be49e2509c9f@52.78.36.216:26656,dd4a3f1750af5765266231b9d8ac764599921736@3.36.224.80:26656,8ea4f592ad6cc38d7532aff418d1fb97052463af@34.240.245.39:26656,e - Bootnodes: bash Mainnet: bootnode ["enode://b8f1cc9c5d4403703fbf377116469667d2b1823c0daf16b7250aa576bacf399e42c3930ccfcb02c5df6879565a2b8931335565f0e8d3f8e72385ecf4a4bf160a@3.36.224.80:30303", "enode://8729e0c825f3d9cad382555f3e46dcff21af323e89025a0e6312df541f4a9e73abfa562d64906f5e59c51fe6f0501b3e61b07979606c56329c020ed739910759@54.194.245.5:30303"] - To check if Heimdall is synced - On the remote machine/VM, run curl localhost:26657/status - In the output, catchingup value should be false - Once Heimdall is synced, run: - sudo service bor start If you've reached this point, you have successfully set up a full node with Ansible. !!! note If Bor presents an error of permission to data, run this command to make the Bor user the owner of the Bor files: bash sudo chown bor /var/lib/bor Logs Logs can be managed by the journalctl linux tool. Here is a tutorial for advanced usage: How To Use Journalctl to View and Manipulate Systemd Logs. Check Heimdall node logs bash journalctl -u heimdalld.service -f Check Bor node logs bash journalctl -u bor.service -f # full-node-binaries.md: This deployment guide walks you through starting and running a full node through various methods. For the system requirements, see the minimum technical requirements guide. !!! tip "Snapshots" Steps in these guide involve waiting for the Heimdall and Bor services to fully sync. This process takes several days to complete. Please use snapshots for faster syncing without having to sync over the network. For detailed instructions, see Sync node using snapshots. For snapshot download links, see the Polygon Chains Snapshots page. Overview !!! warning It is essential to follow the outlined sequence of actions precisely, as any deviation may lead to potential issues. - Prepare the machine. - Install Heimdall and Bor binaries on the full node machine. - Set up Heimdall and Bor services on the full node machine. - Configure the full node machine. - Start the full node machine. - Check node health with the community. Install build-essential This is required for your full node. In order to install, run the below command: bash sudo apt-get update sudo apt-get install build-essential Install binaries Polygon node consists of 2 layers: Heimdall and Bor. Heimdall is a Tendermint fork that monitors contracts in parallel with the Ethereum network. Bor is basically a Geth fork that generates blocks shuffled by Heimdall nodes. Both binaries must be installed and run in the correct order to function properly. Heimdall Install the latest version of Heimdall and related services. Make sure you checkout to the correct release version. To install Heimdall, run the following commands: bash curl -L https://raw.githubusercontent.com/maticnetwork/install/main/heimdall.sh | bash -s -- You can run the above command with following options: - heimdallversion: Valid v1.0+ release tag from https://github.com/maticnetwork/heimdall/releases - networktype: mainnet and amoy - nodetype: sentry This will install the heimdalld and heimdallcli binaries. Verify the installation by checking the Heimdall version on your machine: bash heimdalld version --long Configure Heimdall seeds (Mainnet) bash sed -i 's|^seeds =.|seeds = "1500161dd491b67fb1ac81868952be49e2509c9f@52.78.36.216:26656,dd4a3f1750af5765266231b9d8ac764599921736@3.36.224.80:26656,8ea4f592ad6cc38d7532aff418d1fb97052463af@34.240.245.39:26656,e772e1fb /var/lib/heimdall/config/config.toml chown heimdall /var/lib/heimdall Configure Heimdall seeds (Amoy) The Heimdall and Bor seeds don't need to be configured manually for Amoy testnet since they've already been included at genesis. Bor Install the latest version of Bor, based on valid v1.0+ released version. bash curl -L https://raw.githubusercontent.com/maticnetwork/install/main/bor.sh | bash -s -- You can run the above command with following options: - borversion: valid v1.0+ release tag from https://github.com/maticnetwork/bor/releases - networktype: mainnet and amoy - nodetype: sentry That will install the bor binary. Verify the installation by checking the Bor version on your machine: bash bor version Configure Bor seeds (mainnet) bash sed -i 's|.\[p2p.discovery\]| \[p2p.discovery\] |g' /var/lib/bor/config.toml sed -i 's|.bootnodes =.| bootnodes = ["enode://b8f1cc9c5d4403703fbf377116469667d2b1823c0daf16b7250aa576bacf399e42c3930ccfcb02c5df6879565a2b8931335565f0e8d3f8e72385ecf4a4bf160a@3.36.224.80:30303", "enode://8729e0c825f3d9cad382555f3e46dcff21af323e89025a0e6312df541f4a9e73abfa562d64906f5e59c51fe6f0501b3e61b07979606c56329c020ed739910759@54.194.245.5:30303"]|g' /var/lib/bor/config.toml chown bor /var/lib/bor Configure Bor seeds (Amoy) The Heimdall and Bor seeds don't need to be configured manually for Amoy testnet since they've already been included at genesis. Update service config user permission bash sed -i 's/User=heimdall/User=root/g' /lib/systemd/system/heimdalld.service sed -i 's/User=bor/User=root/g' /lib/systemd/system/bor.service Start services Run the full Heimdall node with these commands on your Sentry Node: bash sudo service heimdalld start !!! warning "Wait for Heimdall to complete syncing" Ensure that Heimdall is fully synced before starting Bor. Initiating Bor without complete synchronization of Heimdall may lead to frequent issues. To check if Heimdall is synced: 1. On the remote machine/VM, run curl localhost:26657/status. 2. In the output, catchingup value should be false. Once Heimdall is synced, run the following command: bash sudo service bor start Logs Logs can be managed by the journalctl linux tool. Here is a tutorial for advanced usage: How To Use Journalctl to View and Manipulate Systemd Logs. Check Heimdall node logs bash journalctl -u heimdalld.service -f Check Bor node logs bash journalctl -u bor.service -f # full-node-docker.md: The Polygon team distributes official Docker images which can be used to run nodes on the Polygon PoS mainnet. These instructions are for running a full Node, but they can be adapted for running sentry nodes and validators as well. Initial setup To get started, you'll need to have shell access with root privileges to a linux machine. !img Install Docker It is likely that your operating system wonâ€™t have Docker installed by default. Please follow the instructions for your particular distribution found here: https://docs.docker.com/engine/install/ Weâ€™re following the instructions for Ubuntu. The steps are included below, but please refer to the official instructions in case theyâ€™ve been updated. bash sudo apt-get update sudo apt-get install ca-certificates curl gnupg lsb-release sudo mkdir -p /etc/apt/keyrings curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg echo \ "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \ $(lsbrelease -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null sudo apt-get update sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin At this point you should have Docker installed. In order to verify, you should be able to run the following command successfully: bash sudo docker run hello-world !img In many cases, itâ€™s inconvenient to run docker as root user so weâ€™ll follow the post install steps here in order to interact with docker without needing to be root: bash sudo groupadd docker sudo usermod -aG docker $USER Now you should be able to logout and log back in, and run docker commands without sudo. Disk setup The exact steps required here may vary a lot based on your needs. Most likely youâ€™ll have a root partition running your operating system on one device. Youâ€™ll probably want one or more devices for actually holding the blockchain data. For the rest of the walkthrough, weâ€™re going to have that additional device mounted at /mnt/data. In this example, we have a device with 4 TB of available space located at /dev/nvme1n1. We are going to mount that using the steps below: bash sudo mkdir /mnt/data sudo mount /dev/nvme1n1 /mnt/data We use df -h to make sure the mount looks good. !img Once we've verified that successfully, we might as well create the home directories on this mount for Bor and Heimdall. bash sudo mkdir /mnt/data/bor sudo mkdir /mnt/data/heimdall Depending on your use case and operating system, youâ€™ll likely want to create an entry in /etc/fstab in order to make sure your device is mounted when the system reboots. In our case we're following some steps like this: bash Use blkid to get the UUID for the device that we're mounting blkid Edit the fstab file and add a line to mount your device UUID={your uuid} /mnt/data {your filesystem} defaults 0 1 sudo emacs /etc/fstab use this to verify the fstab actually works sudo findmnt --verify --verbose At this point you should be able to reboot and confirm that the system loads your mount properly. Heimdall setup At this point, we have a host with docker running on it and we have ample mounted storage to run our Polygon node software. So letâ€™s get Heimdall configured and running. First letâ€™s make sure we can run Heimdall with docker. Run the following command: bash docker run -it 0xpolygon/heimdall:1.0.3 heimdallcli version If this is the first time youâ€™ve run Heimdall with docker, it should pull the required image automatically and output the version information. !img If youâ€™d like to check the details of the Heimdall image or find a different tag, you can take a look at the repository on Docker Hub: https://hub.docker.com/repository/docker/0xpolygon/heimdall At this point, letâ€™s run the Heimdall init command to set up our home directory. bash docker run -v /mnt/data/heimdall:/heimdall-home:rw --entrypoint /usr/bin/heimdalld -it 0xpolygon/heimdall:1.0.3 init --home=/heimdall-home Letâ€™s break this command down a bit in case anything goes wrong. Weâ€™re using docker run to run a command via docker. The switch -v /mnt/data/heimdall:/heimdall-home:rw is very important. Itâ€™s mounting the folder that we created earlier /mnt/data/heimdall from our host system to /heimdall-home within the container as a docker volume. The rw allows the command to write to this docker volume. For all intents and purposes, from within the docker container, the home directory for Heimdall will be /heimdall-home. The argument --entrypoint /usr/bin/heimdalld is overriding the default entry point for this container. The switch it is used to run the command interactively. Finally weâ€™re specifying which image we want to run with 0xpolygon/heimdall:1.0.3. After that init --home=/heimdall-home are arguments being passed to the heimdalld executable. init is the command we want to run and --home is used to specify the location of the home directory. After running the init command, your /mnt/data/heimdall directory should have some structure and look like this: !img Now we need to make a few updates before starting Heimdall. First weâ€™re going to edit the config.toml file. bash Open the config.toml and and make three edits moniker = "YOUR NODE NAME HERE" laddr = "tcp://0.0.0.0:26657" seeds = "LATEST LIST OF SEEDS" sudo emacs /mnt/data/heimdall/config/config.toml If you donâ€™t have a list of seeds, you can find one in this section. In our case, our file has these three lines: A custom human readable name for this node moniker="examplenode01" TCP or UNIX socket address for the RPC server to listen on laddr = "tcp://0.0.0.0:26657" Comma separated list of seed nodes to connect to seeds="f4f605d60b8ffaaf15240564e58a81103510631c@159.203.9.164:26656,4fb1bc820088764a564d4f66bba1963d47d82329@44.232.55.71:26656,2eadba4be3ce47ac8db0a3538cb923b57b41c927@35.199.4.13:26656, !!! warning There are two laddr parameters inside config.toml file. Make sure that you only change the laddr parameter under [rpc]. Now that your config.toml file is all set, youâ€™ll need to make two small changes to your heimdall-config.toml file. Use your favorite editor to update these two settings: RPC endpoint for ethereum chain ethrpcurl = "http://localhost:9545" RPC endpoint for bor chain borrpcurl = "http://localhost:8545" The ethrpcurl should be updated to whatever URL you use for Ethereum mainnet RPC. The borrpcurl in our case is going to be updated to http://bor:8545. After making the edits, our file has these lines: RPC endpoint for ethereum chain ethrpcurl = "https://eth-mainnet.g.alchemy.com/v2/ydmGjsREDACTEDDONTUSE9t7FSf" RPC endpoint for bor chain borrpcurl = "http://bor:8545" The default init command provides a genesis.json but that will not work with PoS mainnet or Amoy. If youâ€™re setting up a mainnet node, you can run this command to download the correct genesis file: bash sudo curl -o /mnt/data/heimdall/config/genesis.json https://raw.githubusercontent.com/maticnetwork/heimdall/master/builder/files/genesis-mainnet-v1.json If you want to verify that you have the right file, you can check against this hash: sha256sum genesis.json 498669113c72864002c101f65cd30b9d6b159ea2ed4de24169f1c6de5bcccf14 genesis.json Starting Heimdall Before we start Heimdall, weâ€™re going to create a docker network so that the containers can easily network with each other based on names. In order to create the network, run the following command: bash docker network create polygon Now weâ€™re going to start Heimdall. Run the following command: bash docker run -p 26657:26657 -p 26656:26656 -v /mnt/data/heimdall:/heimdall-home:rw --net polygon --name heimdall --entrypoint /usr/bin/heimdalld -d --restart unless-stopped 0xpolygon/heimdall:1.0.3 start --home=/heimdall-home Many of the options in this command will look familiar. So letâ€™s talk about whatâ€™s new. The -p 26657:26657 and -p 26656:26656 switches are port mappings. This will instruct docker to map the host port 26657 to the container port 26657 and the same for 26656. The --net polygon switch is telling docker to run this container in the polygon network. --name heimdall is naming the container which is useful for debugging, but itâ€™s all the name that will be used for other containers to connect to Heimdall. The -d argument tells docker to run this container in the background. The switch --restart unless-stopped tells docker to automatically restart the container unless it was stopped manually. Finally, start is being used to actually run the application instead of init which just set up the home directory. At this point itâ€™s helpful to check and see whatâ€™s going on. These two commands can be useful: bash ps will list the running docker processes. At this point you should see one container running docker ps This command will print out the logs directly from the heimdall application docker logs -ft heimdall At this point, Heimdall should start syncing. When you look at the logs, you should see a log of information being spit out that looks like this: 2022-12-14T19:43:23.687640820Z INFO [2022-12-14|19:43:23.687] Executed block module=state height=26079 validTxs=0 invalidTxs=0 2022-12-14T19:43:23.721220869Z INFO [2022-12-14|19:43:23.721] Committed state module=state height=26079 txs=0 appHash=CAEC4C181C9F82D7F55C4BB8A7F564D69A41295A3B62DDAA45F2BB41333DC20F 2022-12-14T19:43:23.730533414Z INFO [2022-12-14|19:43:23.730] Executed block module=state height=26080 validTxs=0 invalidTxs=0 2022-12-14T19:43:23.756646938Z INFO [2022-12-14|19:43:23.756] Committed state module=state height=26080 txs=0 appHash=CAEC4C181C9F82D7F55C4BB8A7F564D69A41295A3B62DDAA45F2BB41333DC20F 2022-12-14T19:43:23.768129711Z INFO [2022-12-14|19:43:23.767] Executed block module=state height=26081 validTxs=0 invalidTxs=0 2022-12-14T19:43:23.794323918Z INFO [2022-12-14|19:43:23.794] Committed state module=state height=26081 txs=0 appHash=CAEC4C181C9F82D7F55C4BB8A7F564D69A41295A3B62DDAA45F2BB41333DC20F 2022-12-14T19:43:23.802989809Z INFO [2022-12-14|19:43:23.802] Executed block module=state height=26082 validTxs=0 invalidTxs=0 2022-12-14T19:43:23.830960386Z INFO [2022-12-14|19:43:23.830] Committed state module=state height=26082 txs=0 appHash=CAEC4C181C9F82D7F55C4BB8A7F564D69A41295A3B62DDAA45F2BB41333DC20F 2022-12-14T19:43:23.840941976Z INFO [2022-12-14|19:43:23.840] Executed block module=state height=26083 validTxs=0 invalidTxs=0 2022-12-14T19:43:23.866564767Z INFO [2022-12-14|19:43:23.866] Committed state module=state height=26083 txs=0 appHash=CAEC4C181C9F82D7F55C4BB8A7F564D69A41295A3B62DDAA45F2BB41333DC20F 2022-12-14T19:43:23.875395744Z INFO [2022-12-14|19:43:23.875] Executed block module=state height=26084 validTxs=0 invalidTxs=0 If youâ€™re not seeing any information like this, your node might not be finding enough peers. The other useful command at this point is an RPC call to check the status of Heimdall syncing: bash curl localhost:26657/status This will return a response like: json { "jsonrpc": "2.0", "id": "", "result": { "nodeinfo": { "protocolversion": { "p2p": "7", "block": "10", "app": "0" }, "id": "0698e2f205de0ffbe4ca215e19b2ee7275d2c334", "listenaddr": "tcp://0.0.0.0:26656", "network": "heimdall-137", "version": "0.32.7", "channels": "4020212223303800", "moniker": "examplenode01", "other": { "txindex": "on", "rpcaddress": "tcp://0.0.0.0:26657" } }, "syncinfo": { "latestblockhash": "812700055F33B175CF90C870B740D01B0C5B5DCB8D22376D2954E1859AF30458", "latestapphash": "83A1568E85A1D942D37FE5415F3FB3CBD9DFD846A42CBC247DFD6ABB9CE7E606", "latestblockheight": "16130", "latestblocktime": "2020-05-31T17:06:31.350723885Z", "catchingup": true }, "validatorinfo": { "address": "3C6058AF387BB74D574582C2BEEF377E7A4C0238", "pubkey": { "type": "tendermint/PubKeySecp256k1", "value": "BOIKA6z1q3l5iSJoaAiagWpwUw3taAhiEMyZ9ffxAMznas2GU1giD5YmtnrB6jzp4kkIqv4tOmuGYILSdy9+vYI=" }, "votingpower": "0" } } } In this initial setup phase, itâ€™s important to pay attention to the syncinfo field. If catchingup is true, it means that Heimdall is not fully synced. You can check the other properties within syncinfo to get a sense how far behind Heimdall is. Starting Bor At this point, you should have a node thatâ€™s successfully running Heimdall. You should be ready now to run Bor. Before we get started with Bor, we need to run the Heimdall rest server. This command will start a REST API that Bor uses to retrieve information from Heimdall. The command to start the server is: bash docker run -p 1317:1317 -v /mnt/data/heimdall:/heimdall-home:rw --net polygon --name heimdallrest --entrypoint /usr/bin/heimdalld -d --restart unless-stopped 0xpolygon/heimdall:1.0.3 rest-server --home=/heimdall-home --node "tcp://heimdall:26657" There are two options used in this command that are different and worth noting. Rather than running the start command, weâ€™re running the rest-server command. Also, weâ€™re passing â€“node â€œtcp://heimdall:26657â€ which tells the rest server how to communicate with Heimdall. If this command runs successfully, when you run docker ps, you should see two commands containers running now. Additionally, if you run this command you should see some basic output: bash curl localhost:1317/bor/span/1 Bor will rely on this interface. So if you donâ€™t see JSON output, there is something wrong! Now letâ€™s download the genesis file for Bor specifically: bash sudo curl -o /mnt/data/bor/genesis.json 'https://raw.githubusercontent.com/maticnetwork/bor/master/builder/files/genesis-mainnet-v1.json' Letâ€™s verify the sha256 sum again for this file: sha256sum genesis.json 4bacbfbe72f0d966412bb2c19b093f34c0a1bd4bb8506629eba1c9ca8c69c778 genesis.json Now we need to create a default config file for starting Bor. bash docker run -it 0xpolygon/bor:1.1.0 dumpconfig | sudo tee /mnt/data/bor/config.toml This command is going to generate a .toml file with default settings. Weâ€™re going to make a few changes to the file, so open it up with your favorite editor and make a few updates. Note: weâ€™re only showing the lines that are changed. For reference, you can see the details for the Bor image here: https://hub.docker.com/repository/docker/0xpolygon/bor bash Similar to moniker, you might want to update this with a name of your own choosing identity = "docker.example" Setting this to the location of a mount that we'll make datadir = "/bor-home" We'll want to specify some boot nodes [p2p] [pep.discovery] bootnodes = ["enode://0cb82b395094ee4a2915e9714894627de9ed8498fb881cec6db7c65e8b9a5bd7f2f25cc84e71e89d0947e51c76e85d0847de848c7782b13c0255247a6758178c@44.232.55.71:30303", "enode://88116f4295f5a31538ae409e4d44ad40d22e44ee9342869e7d68bdec55b0f83c1530355ce8b41fbec0928a7d75a5745d528450d30aec92066ab6ba1ee351d710@159.203.9.164:30303"] Because we're running inside docker, we'll likely need to change the way we connect to heimdall [heimdall] url = "http://heimdallrest:1317" Assuming you want to access the RPC, you'll need to make a change here as well [jsonrpc] [jsonrpc.http] enabled = true host = "0.0.0.0" At this point, we should be ready to start Bor. Weâ€™re going to use this command: bash docker run -p 30303:30303 -p 8545:8545 -v /mnt/data/bor:/bor-home:rw --net polygon --name bor -d --restart unless-stopped 0xpolygon/bor:1.1.0 server --config /bor-home/config.toml If everything goes well, you should see log entries that look like this: bash 2022-12-14T19:53:51.989897291Z INFO [12-14|19:53:51.989] Fetching state updates from Heimdall fromID=4 to=2020-05-30T23:47:46Z 2022-12-14T19:53:51.989925064Z INFO [12-14|19:53:51.989] Fetching state sync events queryParams="from-id=4&to-time=1590882466&limit=50" 2022-12-14T19:53:51.997640841Z INFO [12-14|19:53:51.997] StateSyncData Gas=0 Block-number=12800 LastStateID=3 TotalRecords=0 2022-12-14T19:53:52.021990622Z INFO [12-14|19:53:52.021] Fetching state updates from Heimdall fromID=4 to=2020-05-30T23:49:58Z 2022-12-14T19:53:52.022015930Z INFO [12-14|19:53:52.021] Fetching state sync events queryParams="from-id=4&to-time=1590882598&limit=50" 2022-12-14T19:53:52.040660857Z INFO [12-14|19:53:52.040] StateSyncData Gas=0 Block-number=12864 LastStateID=3 TotalRecords=0 2022-12-14T19:53:52.064795784Z INFO [12-14|19:53:52.064] Fetching state updates from Heimdall fromID=4 to=2020-05-30T23:52:10Z 2022-12-14T19:53:52.064828634Z INFO [12-14|19:53:52.064] Fetching state sync events queryParams="from-id=4&to-time=1590882730&limit=50" 2022-12-14T19:53:52.085029612Z INFO [12-14|19:53:52.084] StateSyncData Gas=0 Block-number=12928 LastStateID=3 TotalRecords=0 2022-12-14T19:53:52.132067703Z INFO [12-14|19:53:52.131] âœ… Committing new span id=3 startBlock=13056 endBlock=19455 validatorBytes=f8b6d906822710940375b2fc7140977c9c76d45421564e354ed42277d9078227109442eefcda06ead475cde3731b8eb138e88cd0bac3d9018238a2945973918275c01f50555d44e92c9d9b353cadad54d9058227 producerBytes=f8b6d906822710940375b2fc7140977c9c76d45421564e354ed42277d9078227109442eefcda06ead475cde3731b8eb138e88cd0bac3d9018238a2945973918275c01f50555d44e92c9d9b353cadad54d9058227 2022-12-14T19:53:52.133545235Z INFO [12-14|19:53:52.133] Fetching state updates from Heimdall fromID=4 to=2020-05-30T23:54:22Z 2022-12-14T19:53:52.133578948Z INFO [12-14|19:53:52.133] Fetching state sync events queryParams="from-id=4&to-time=1590882862&limit=50" 2022-12-14T19:53:52.135049605Z INFO [12-14|19:53:52.134] StateSyncData Gas=0 Block-number=12992 LastStateID=3 TotalRecords=0 2022-12-14T19:53:52.152067646Z INFO [12-14|19:53:52.151] Fetching state updates from Heimdall fromID=4 to=2020-05-30T23:56:34Z 2022-12-14T19:53:52.152198357Z INFO [12-14|19:53:52.151] Fetching state sync events queryParams="from-id=4&to-time=1590882994&limit=50" 2022-12-14T19:53:52.176617455Z INFO [12-14|19:53:52.176] StateSyncData Gas=0 Block-number=13056 LastStateID=3 TotalRecords=0 2022-12-14T19:53:52.191060112Z INFO [12-14|19:53:52.190] Fetching state updates from Heimdall fromID=4 to=2020-05-30T23:58:46Z 2022-12-14T19:53:52.191083740Z INFO [12-14|19:53:52.190] Fetching state sync events queryParams="from-id=4&to-time=1590883126&limit=50" 2022-12-14T19:53:52.223836639Z INFO [12-14|19:53:52.223] StateSyncData Gas=0 Block-number=13120 LastStateID=3 TotalRecords=0 2022-12-14T19:53:52.236025906Z INFO [12-14|19:53:52.235] Fetching state updates from Heimdall fromID=4 to=2020-05-31T00:00:58Z 2022-12-14T19:53:52.236053406Z INFO [12-14|19:53:52.235] Fetching state sync events queryParams="from-id=4&to-time=1590883258&limit=50" 2022-12-14T19:53:52.269611566Z INFO [12-14|19:53:52.269] StateSyncData Gas=0 Block-number=13184 LastStateID=3 TotalRecords=0 2022-12-14T19:53:52.283199351Z INFO [12-14|19:53:52.283] Fetching state updates from Heimdall fromID=4 to=2020-05-31T00:03:10Z 2022-12-14T19:53:52.283737573Z INFO [12-14|19:53:52.283] Fetching state sync events queryParams="from-id=4&to-time=1590883390&limit=50" 2022-12-14T19:53:52.314141359Z INFO [12-14|19:53:52.314] StateSyncData Gas=0 Block-number=13248 LastStateID=3 TotalRecords=0 2022-12-14T19:53:52.325150782Z INFO [12-14|19:53:52.325] Fetching state updates from Heimdall fromID=4 to=2020-05-31T00:05:22Z 2022-12-14T19:53:52.325171075Z INFO [12-14|19:53:52.325] Fetching state sync events queryParams="from-id=4&to-time=1590883522&limit=50" 2022-12-14T19:53:52.354470271Z INFO [12-14|19:53:52.354] StateSyncData Gas=0 Block-number=13312 LastStateID=3 TotalRecords=0 2022-12-14T19:53:52.372354857Z INFO [12-14|19:53:52.372] Fetching state updates from Heimdall fromID=4 to=2020-05-31T00:07:34Z 2022-12-14T19:53:52.372389214Z INFO [12-14|19:53:52.372] Fetching state sync events queryParams="from-id=4&to-time=1590883654&limit=50" 2022-12-14T19:53:52.398246950Z INFO [12-14|19:53:52.398] StateSyncData Gas=0 Block-number=13376 LastStateID=3 TotalRecords=0 2022-12-14T19:53:52.413321099Z INFO [12-14|19:53:52.413] Fetching state updates from Heimdall fromID=4 to=2020-05-31T00:09:46Z 2022-12-14T19:53:52.413345355Z INFO [12-14|19:53:52.413] Fetching state sync events queryParams="from-id=4&to-time=1590883786&limit=50" 2022-12-14T19:53:52.437176855Z INFO [12-14|19:53:52.437] StateSyncData Gas=0 Block-number=13440 LastStateID=3 TotalRecords=0 2022-12-14T19:53:52.450356966Z INFO [12-14|19:53:52.450] Fetching state updates from Heimdall fromID=4 to=2020-05-31T00:11:58Z There are a few ways to check the sync state of Bor. The simplest is using curl: bash curl 'localhost:8545/' \ --header 'Content-Type: application/json' \ -d '{ "jsonrpc":"2.0", "method":"ethsyncing", "params":[], "id":1 }' When you run this command, you'll see an output like this: json { "jsonrpc": "2.0", "id": 1, "result": { "currentBlock": "0x2eebf", "healedBytecodeBytes": "0x0", "healedBytecodes": "0x0", "healedTrienodeBytes": "0x0", "healedTrienodes": "0x0", "healingBytecode": "0x0", "healingTrienodes": "0x0", "highestBlock": "0x1d4ee3e", "startingBlock": "0x0", "syncedAccountBytes": "0x0", "syncedAccounts": "0x0", "syncedBytecodeBytes": "0x0", "syncedBytecodes": "0x0", "syncedStorage": "0x0", "syncedStorageBytes": "0x0" } } This will indicate the currentBlock thatâ€™s been synced and also the highestBlock that weâ€™re aware of. If the node is already synced, we should get false. Seed nodes and bootnodes !!! info "Amoy testnet seeds" The Bor and Heimdall seeds don't need to be configured manually for Amoy testnet since they've already been included at genesis. Heimdall seed nodes bash Mainnet: seeds = "1500161dd491b67fb1ac81868952be49e2509c9f@52.78.36.216:26656,dd4a3f1750af5765266231b9d8ac764599921736@3.36.224.80:26656,8ea4f592ad6cc38d7532aff418d1fb97052463af@34.240.245.39:26656,e772e1fb !!! tip The following Heimdall seed can also be used for mainnet: 8542cd7e6bf9d260fef543bc49e59be5a3fa9074@seed.publicnode.com:27656 Bootnodes bash Mainnet: bootnodes = ["enode://b8f1cc9c5d4403703fbf377116469667d2b1823c0daf16b7250aa576bacf399e42c3930ccfcb02c5df6879565a2b8931335565f0e8d3f8e72385ecf4a4bf160a@3.36.224.80:30303", "enode://8729e0c825f3d9cad382555f3e46dcff21af323e89025a0e6312df541f4a9e73abfa562d64906f5e59c51fe6f0501b3e61b07979606c56329c020ed739910759@54.194.245.5:30303"] # full-node-gcp.md: This document describes how to deploy Polygon PoS nodes in a virtual machine (VM) instance on the Google Cloud Platform (GCP). It is recommended to use any modern Debian or Linux Ubuntu OS with long-term support, i.e. Debian 11, Ubuntu 20.04. We'll focus on Ubuntu 20.04 in this guide. !!! info This setup is currently only supported for mainnet. GCP support for deploying Amoy testnet nodes will be available soon. Deploy VM instance You may use any of the following ways to create an instance in Google Cloud: 1. Google Cloud CLI, local or Cloud Shell 2. Web Console We only cover the first case in this guide. Let's start with deployment using Google Cloud CLI. 1. Follow "Before you begin" section to install and configure gcloud command-line tool. Pay attention to default region and zone, choose ones closer to you or your customers. You may use gcping.com to measure latency to choose the closest location. 2. Adjust the following command variables using your favorite editor prior to executing, when required: POLYGONNETWORK - choose mainnet network to run. POLYGONNODETYPE - choose archive,fullnode node type to run. POLYGONBOOTSTRAPMODE - choose bootstrap mode snapshot or fromscratch. POLYGONRPCPORT - choose JSON RPC bor node port to listen on, the default value is what used on VM instance creation and in firewall rules. EXTRAVAR - choose Bor and Heimdall branches, use networkversion=mainnet-v1 with mainnet network and networkversion=testnet-v4 with amoy network. INSTANCENAME - the name of a VM instance with Polygon we are going to create. INSTANCETYPE - GCP machine type, default value is recommended, You may change it later if required. BOREXTDISKSIZE - additional disk size in GB to use with Bor, default value with fullnode is recommended, You may expand it later if required. You'll need 8192GB+ with archive node though. HEIMDALLEXTDISKSIZE - additional disk size in GB to use with Heimdall, default value is recommended. DISKTYPE - GCP disk type, SSD is highly recommended. You may need to increase the total SSD GB quota in the region you are spinning up the node. 3. Use the following command to create an instance with the correct hardware and software requirements. In the example below, we deploy Polygon PoS mainnet from snapshot in the fullnode mode: bash export POLYGONNETWORK=mainnet export POLYGONNODETYPE=fullnode export POLYGONBOOTSTRAPMODE=snapshot export POLYGONRPCPORT=8747 export GCPNETWORKTAG=polygon export EXTRAVAR=(borbranch=v1.1.0 heimdallbranch=v1.0.3 networkversion=mainnet-v1 nodetype=sentry/sentry heimdallnetwork=${POLYGONNETWORK}) gcloud compute firewall-rules create "polygon-p2p" --allow=tcp:26656,tcp:30303,udp:30303 --description="polygon p2p" --target-tags=${GCPNETWORKTAG} gcloud compute firewall-rules create "polygon-rpc" --allow=tcp:${POLYGONRPCPORT} --description="polygon rpc" --target-tags=${GCPNETWORKTAG} export INSTANCENAME=polygon-0 export INSTANCETYPE=e2-standard-8 export BOREXTDISKSIZE=1024 export HEIMDALLEXTDISKSIZE=500 export DISKTYPE=pd-ssd gcloud compute instances create ${INSTANCENAME} \ --image-project=ubuntu-os-cloud \ --image-family=ubuntu-2004-lts \ --boot-disk-size=20 \ --boot-disk-type=${DISKTYPE} \ --machine-type=${INSTANCETYPE} \ --create-disk=name=${INSTANCENAME}-bor,size=${BOREXTDISKSIZE},type=${DISKTYPE},auto-delete=no \ --create-disk=name=${INSTANCENAME}-heimdall,size=${HEIMDALLEXTDISKSIZE},type=${DISKTYPE},auto-delete=no \ --tags=${GCPNETWORKTAG} \ --metadata=user-data=' #cloud-config bootcmd: - screen -dmS polygon su -l -c bash -c "curl -L https://raw.githubusercontent.com/maticnetwork/node-ansible/master/install-gcp.sh | bash -s -- -n '${POLYGONNETWORK}' -m '${POLYGONNODETYPE}' -s '${POLYGONBOOTSTRAPMODE}' -p '${POLYGONRPCPORT}' -e \"'${EXTRAVAR}'\"; bash"' The instance should be created and live in a couple of minutes. Login to VM It will take a couple of minutes to install all the required software, and a couple of hours to download a snapshot when chosen. - You should see working bor and heimdalld processes filling up additional drives. You may run the following commands to check it. bash gcloud compute ssh ${INSTANCENAME} inside the connected session sudo su - ps uax|egrep "bor|heimdalld" df -l -h - You may use the following command to watch the installation progress, it's really handy in case of snapshot bootstrap: bash inside the connected session screen -dr Use Control+a d key combination to disconnect from progress review. - You may use the following commands to get Bor and Heimdall logs: bash inside the connected session journalctl -fu bor journalctl -fu heimdalld !!! info Blockchain data is saved onto additional drives which are kept by default on VM instance removal. You need to remove additional disks manually if you don't need this data anymore. At the end, you will get an instance as shown in the diagram below. !Figure: Mainnet - Polygon instance # full-node-packages.md: Overview - Prepare the Full Node machine. - Install Heimdall and Bor packages on the Full Node machine. - Configure the Full node. - Start the Full node. !!! warning It is essential to follow the outlined sequence of actions precisely, as any deviation may lead to potential issues. Install packages Prerequisites - One machine is needed. - Bash is installed on the machine. Heimdall - Install the default latest version of sentry for Mainnet: shell curl -L https://raw.githubusercontent.com/maticnetwork/install/main/heimdall.sh | bash -s -- or install a specific version, node type (sentry or validator), and network (mainnet or amoy). All release versions can be found on Heimdall GitHub repository. shell Example: curl -L https://raw.githubusercontent.com/maticnetwork/install/main/heimdall.sh | bash -s -- v1.0.7 mainnet sentry Bor - Install the default latest version of sentry for Mainnet: shell curl -L https://raw.githubusercontent.com/maticnetwork/install/main/heimdall.sh | bash -s -- or install a specific version, node type (sentry or validator), and network (mainnet or amoy). All release versions could be found on Bor Github repository. shell Example: curl -L https://raw.githubusercontent.com/maticnetwork/install/main/bor.sh | bash -s -- v1.3.7 mainnet sentry Configuration Configure Heimdall - Initialize Heimdall configs shell For mainnet sudo -u heimdall heimdalld init --chain=mainnet --home /var/lib/heimdall For testnet sudo -u heimdall heimdalld init --chain=amoy --home /var/lib/heimdall - You will need to add a few details in the config.toml file. To open the config.toml file run the following command vi /var/lib/heimdall/config/config.toml - Now in the config file you will have to change Moniker shell moniker= For example, moniker=my-sentry-node - Change the value of Prometheus to true - Set the maxopenconnections value to 100 Make sure you keep the proper formatting when you make the changes above. Configure service files for Bor and Heimdall After successfully installing Bor and Heimdall through packages, their service file could be found under /lib/systemd/system, and Bor's config file could be found under /var/lib/bor/config.toml. You will need to check and modify these files accordingly. - Make sure the chain is set correctly in /lib/systemd/system/heimdalld.service file. Open the file with following command sudo vi /lib/systemd/system/heimdalld.service - In the service file, set --chain to mainnet or amoy accordingly Save the changes in /lib/systemd/system/heimdalld.service. - Make sure the chain is set correctly in /var/lib/bor/config.toml file. Open the file with following command sudo vi /var/lib/bor/config.toml - In the config file, set chain to mainnet or amoy accordingly. - To enable Archive mode you can optionally enable the following flags: js gcmode "archive" [jsonrpc] [jsonrpc.ws] enabled = true port = 8546 corsdomain = ["*"] Save the changes in /var/lib/bor/config.toml. Start services Reloading service files to make sure all changes to service files are loaded correctly. shell sudo systemctl daemon-reload Start Heimdall, Heimdall rest server, and Heimdall bridge. shell sudo service heimdalld start You can also check Heimdall logs with the following command: shell journalctl -u heimdalld.service -f !!! warning At this point, please make sure that Heimdall is synced completely, and only then start Bor. If you start Bor without Heimdall syncing completely, you will run into issues frequently. To check if Heimdall is synced: - On the remote machine/VM, run curl localhost:26657/status - In the output, catchingup value should be false Now, once Heimdall is synced, run: shell sudo service bor start You can check Bor logs using the following command: shell journalctl -u bor.service -f # change-signer-address.md: This guide will walk you through the process of changing the owner and signer address for your validator node. Change the owner address 1. Access StakingNFT smart contract. - Mainnet: https://etherscan.io/address/0x47Cbe25BbDB40a774cC37E1dA92d10C2C7Ec897F#writeContract - Amoy: https://sepolia.etherscan.io/address/0x72CF5618142Eb369E75ec6529A907e9A6Fe99bB7#writeContract 2. Click on the Connect to Web3 button and login using the owner address of your validator node.

!change-owner-1{width=30%}

3. You will see a list of functions appear. Click on safeTransferFrom function. This will be 5th function in the list. There will be 3 data fields that you will need to add information.

!change-owner-2

4. Here: - from (address) is your current owner address - to (address) is your new owner Address - tokenId is your validator ID 5. Fill in the relevant information and select Write. You will be prompted to sign a transaction. Ensure that you have sufficient ETH to make the transaction. 6. Upon signing the transaction your validator NFT will be transferred to the new owner address. 7. Log in to theÂ staking dashboardÂ with the new owner address to verify the changes. Change the signer address This guide refers to your current validator node as Node 1 and your new validator node as Node 2. 1. Log in to the staking dashboard with the Node 1 address. 2. On your profile, select Edit Profile. 3. In the Signer's address field, enter the Node 2 address. 4. In the Signer's public key field, enter the Node 2 public key. To get the public key, run the following command on the validator node: sh heimdalld show-account Selecting Save will save your new details for your node. This essentially means that Node 1 will be your address that controls the stake, where the rewards will be sent to, etc. And Node 2 will now be performing activities like signing blocks, signing checkpoints, etc. # next-steps.md: !!! info "Limited spots for new validators" There is limited space for accepting new validators. New validators can only join the active set when an already active validator unbonds. Check out the following links for more information and to apply for a validator slot: - Admission form - Admission dashboard Once your validator node is onboarded into the active set, do the following: - Log in to theÂ staking dashboardÂ with theÂ owner address. - Go to my account, and click on edit details below the validator name. - Click on Profile Details and update your name, website, description, logo URL, and click on Save Profile Details. Stake tokens Initial staking 1. Access the validator dashboard. 2. Log in with your wallet. You can use a popular wallet such as MetaMask. Make sure you login using the owner address, and that you have POL tokens in the wallet. 3. Select Become a Validator. You will be asked to set up your node. If you haven't already set up your node by now, you will need to do so, else if you proceed ahead you will receive an error when you attempt to stake. 4. On the next screen, add your validator details, the commission rate, and the staking amount. 5. Select Stake Now. 6. Now, you'll be prompted for three confirmations to send the transaction. Once complete, your POL tokens will be added to the staked amount on the validator node. The three confirmations include: - Approve Transaction: This approves your stake transaction. - Stake: Confirms your stake transaction. - Save: Saves your validator details. !!! info For the changes to take effect on the staking dashboard, it requires a minimum of 12 block confirmations. Add stake 1. Access theÂ validator dashboard. 2. Log in with your wallet. You can use a popular wallet such as MetaMask. Make sure you login using the owner address, and that you have POL tokens in the wallet. 3. SelectÂ Add more

Stake. 4. Enter the amount, and select Add More Stake. 5. Now, you'll be prompted for three confirmations to send the transaction. Once complete, your POL tokens will be added to your staked amount on the validator node. The three confirmations include: - Approve Transaction: This approves your stake transaction. - Stake: Confirms your stake transaction. - Save: Saves your validator details. !!! info For the changes to take effect on the staking dashboard, it requires a minimum of 12 block confirmations. Set commission rate You can set up and change your commission as a validator. A validator is entitled to charge any commission rate. The minimum commission would be 0% and the maximum commission would be 100% of the rewards earned. You set up the commission rate as part of your initial validator staking process. Changing your commission rate You are allowed to freely adjust the commission rate as and when necessary. As a validator, it is one of your responsibilities to inform the community on commission changes. See Validator Responsibilities. Follow the steps below to change your commission rate: 1. With your owner address, login to the staking dashboard. 2. On your profile, select Edit Profile. 3. In the Commission field, enter your new commission rate. Once you have confirmed and signed the transaction your commission rate will be set. Note that once the commission is updated, there is a cool down period of 80 checkpoints. Claim validator rewards Once you are set up and staked as a validator, you will earn rewards for performing validator duties. When you perform validator duties dutifully, you get rewarded. To claim rewards you can go to yourÂ validator dashboard. You will see two buttons on your profile: - Withdraw Reward - Restake Reward Withdraw Reward As a validator, you earn rewards as long as you are performing your validator duties correctly. SelectingÂ Withdraw RewardÂ will get your rewards back to your wallet. The dashboard will update afterÂ 12 block confirmations. Restake Reward Restaking your rewards is an easy way to increase your stake as a validator. SelectingÂ Restake RewardÂ will restake your reward and increase your stake. The dashboard will update afterÂ 12 block confirmations. Common operations You can use the following commands to check if your validator node is set up correctly. Check validator account Run the following command on your validator node to check if the account is set up correctly: sh heimdalld show-account The output should appear in the following format: json { "address": "0x6c468CF8c9879006E22EC4029696E005C2319C9D", "pubkey": "0x04b12d8b2f6e3d45a7ace12c4b2158f79b95e4c28ebe5ad54c439be9431d7fc9dc1164210bf6a5c3b8523528b931e772c86a307e8cff4b725e6b4a77d21417bf19" } This will display your address and public key for your validator node. Note that this address must match with your signer address on Ethereum. Show private key Run the following command on your validator node to check if the private key configured correctly: sh heimdalld show-privatekey The output should appear in the following format: json { "privkey": "0x" } Check the balance To check the balance of your address, run the following command: sh heimdallcli query auth account SIGNERADDRESS --chain-id CHAINID where, SIGNERADDRESS: Your signer address. CHAINID: The Polygon mainnet chain ID with the client prefix: heimdall-137. The following output should appear: json address: 0x6c468cf8c9879006e22ec4029696e005c2319c9d coins: - denom: pol amount: i: "1000000000000000000000" accountnumber: 0 sequence: 0 # topup-heimdall-fee.md: This document provides instructions on how to top up the Heimdall fee for a PoS validator. The easiest way is via the Polygon Staking UI. You can also do it manually by following the steps below. This requires basic Etherscan knowledge and key details like the validator signer address. 1. Head over to Etherscan.io 2. Enter the StakeManagerProxy contract address in the search box: 0x5e3Ef299fDDf15eAa0432E6e66473ace8c13D908 3. Under the Code menu, select the Write as Proxy tab. Connect your Web3 wallet using the Connect to Web3 button. !Figure: Connect wallet etherscan 4. Scroll down to the topUpForFee method (#26 in the list) and select it. You will then see a display similar to the screenshot below.. !Figure: Top up Heimdall fee 5. Fill in the details: - user: Validator's Signer Address - heimdallFee: Top-up fee (minimum 1 POL) 6. After filling in the details, select Write to sign the transaction. Your Heimdall fee will be updated soon after the transaction completes. # validator-performance.md: Validator performance framework This is a guide to help users understand the validator performance parameters the community agreed to in PIP-4. These parameters and the related health statuses help provide information about validator performance against a defined performance benchmark. Parameters A validatorâ€™s performance is measured based on the checkpoints it signed over a fixed monitoring period. Performance is measured on a rolling basis at each new checkpoint to provide an objective figure. This figure is then measured against a benchmark of the total network performance in the monitoring period, as detailed below. Monitoring Period (â€œMP") = previous 700 checkpoints, updated every new checkpoint. Take % of checkpoints signed by each validator in the MP and find the median. Multiply the median average by an agreed multiple = Performance Benchmark (â€œPBâ€). At each checkpoint, calculate the % of checkpoints signed in the MP by single validators and measure against the PB. Performance benchmark

!Figure: performance benchmark{width=60%}

To facilitate the transition, there will be a slightly lower benchmark around the first two months while validators become accustomed to the parameters. PB1 â†' 95% of the median average of the last 700 checkpoints signed by the validator set (first 2,800 checkpoints) PB2 â†' 98% of the median average of last checkpoints signed by validator set (continues thereafter) Deficient validator process If validator !Figure: Health status Public notices The Public Notices page shows the recent notices and messages sent to the community of validators. !Figure: Topup Heimdall fee This is what a notice message should look like:

!Figure: Notice{width=60%}

Forced unstaking The unstaking of the deficient validator would be done as follows: Call the ForceUnstake function in Polygon Commitchain Contract: 0xFa7D2a996aC6350f4b56C043112Da0366a59b74c # known-issues.md: This resource is designed to help you efficiently diagnose and resolve common issues encountered while interacting with the Polygon PoS network as a validator. Bor Bor is unable to connect to peers Bor stops importing new blocks, with logs displaying messages similar to the following: js Aug 19 13:33:35 polygon-mainnet-validator-backup-4 bor[124475]: INFO [08-19|13:33:35.123] Looking for peers peercount=0 tried=0 static=7 Aug 19 13:33:36 polygon-mainnet-validator-backup-4 bor[124475]: INFO [08-19|13:33:36.916] Whitelisting milestone deferred err="chain out of sync" Aug 19 13:33:48 polygon-mainnet-validator-backup-4 bor[124475]: INFO [08-19|13:33:48.916] Whitelisting milestone deferred err="chain out of sync" Solution - Increase maxpeer count to 200 - Add the bootnodes under static and trusted nodes - If this doesnâ€™t resolve the issue, try adding the peers manually using the IPC console Error: Bad block/Invalid Merkle A bad block or invalid Merkle root error occurs when the Heimdall and Bor layers are not in sync. Heimdall, as the consensus layer for Polygon POS chain, directs Bor to create blocks accordingly. A bad block error occurs when the Bor moves ahead to create a block which has not been directed by Heimdall. This causes an invalid hash being created, and hence results in an invalid Merkle root. Solution 1 Restart the Bor service using the following command: bash sudo service bor restart Typically a restart of the Bor service should resolve the problem, and that's because restarting causes Bor to reconnect with Heimdall, start syncing, and create blocks correctly. If restarting the Bor service does not fix the problem, try the next option. Solution 2 Make the following checks: - Check if your Heimdall and REST servers are running. The Heimdall service might have stopped, and thus causing the bad block issue on Bor. - Check the logs for your Heimdall first using the following command: bash journalctl -u heimdalld -f - Check if everything is working correctly. - Restart the services that are not running. This should cause Bor to automatically resolve the problem If restarting both the Bor and Heimdall services doesn't solve the problem, it could be that Bor is stuck on a particular block. Solution 3 Check the bad block in logs for Bor. - Check Bor logs with this command: bash journalctl -u bor -f The bad block is typically displayed in the logs as shown in the below figure: !Figure: Bad block - Note the bad block number. - Convert the block number to a hexadecimal number. !!! info Use this tool to convert the block number to a hexadecimal number. - Roll back the chain by a few hundred blocks, i.e., set Bor at the right block height using the debug.setHead() function. Use the following command: bash bor attach ./.bor/data/bor.ipc > debug.setHead("0xE92570") The debug.setHead() function allows Bor to set the tip at a particular block height, resyncing from a previous block. The command should return a null upon successful execution. Once this occurs, you can resume monitoring Bor to verify whether the chain progresses beyond the previously problematic block. If none of these solutions work for you, please contact the Polygon Support team immediately. Issue: Bor synchronization is slow If Bor synchronization is slow, it may be caused by one or more of the following factors: - The node is running on a fork - means at certain point the block production was done by forking on a different block and that has impacted the further block production. - The machine is not working at optimum levels and could be with insufficient resources. This can be addressed by checking the following: - IOPS - IOPS stands for Input/Output state of cycle. - The rate of reading is usually higher than write speed. - 6000 is the recommended range for IOPS. - Processing power - Processor has to be 8 or 16 core. - RAM: 32 GB is the minimum; 64 GB is recommended. - Block import should be more than 2 block for every second. - Node sync rate should be at 15-20 blocks every 8 secs. Solution Since the issue is likely due to insufficient hardware resources, consider upgrading to double the current specs. Validator Bor is stuck on a block for a long time This implies that the Bor service on your sentry node is also stuck because your validator gets information from your sentry. Solution - Please check the Bor logs on your sentry and see if everything is normal and functional. - Restart the Bor service on your sentry node, then simultaneously restart the Bor service on your validator. Retrying again in 5 seconds to fetch data from Heimdall path=bor/span/1 These logs in Bor mean that it cannot connect to Heimdall. Heimdall appears to be out of sync, and thus it lacks the data needed by Bor. Solution The recommended approach is to clear the historical data from both Heimdall and Bor, then resync using a snapshot. Verify the following: 1. Are Heimdall logs normal, or do they show any errors? 2. Confirm Heimdall is fully synced by running: curl localhost:26657/status 3. Check whether Heimdall is connected to other peers. bash curl localhost:26657/netinfo? | jq .result.npeers If there are no peers, verify that the seeds or persistent peers are correctly configured on Heimdall, and ensure that port 26656 is open. etherbase missing: etherbase must be explicitly specified To fix this issue, the signer address that is used to mine must be added in the miner.etherbaseÂ section in theÂ config.tomlÂ file. Error: Failed to unlock account (0xâ€¦) No key for given address or file This error occurs because the path to the password.txt file is incorrect. Follow the steps below to resolve this issue. Solution 1. Kill the Bor process. 2. Copy the Bor keystore file to: /var/lib/bor/keystore/ 3. And the password.txt file to: /var/lib/bor/password.txt 4. Ensure that the user 'Bor' has permission to access the password.txt file. You can do this by running the following command: sudo chown -R bor:nogroup /var/lib/bor/ Steps to prune the node Follow the steps below to prune your node: 1. Check your Bor data size before pruning using the following command: bash du -sh /usr/bin/bor 2. Stop Bor. bash sudo service bor stop 3. Start tmux to ensure that even if your SSH connection is reset, the process is running on the remote machine using tmux. 4. Start pruning. bash sudo bor snapshot prune-state --datadir /usr/bin/bor The default --datadir is /usr/bin/bor. 5. Once the pruning is completed, you will see success logs and details. Then start Bor again using: bash sudo service bor start 6. Check your Bor data size after pruning using: bash du -sh /usr/bin/bor Heimdall Log: Error dialing seed/Looking for peers or stopping peer for error/Dialing failed This log is expected when you first start Heimdall, as it takes some time to find and connect to peers. If the issue persists, check the following: - Verify that your Heimdall node is configured with the latest seeds as listed in the node setup documentation. If the error persists after updating to the latest seeds or confirming that you are using the correct ones, follow these steps: 1. IncreaseÂ maxnuminboundpeersÂ andÂ maxnumoutboundpeersÂ inÂ /var/lib/heimdall/config/config.toml: toml maxnuminboundpeers = 300 maxnumoutboundpeers = 100 2. StartÂ heimdalldÂ service using the following command: bash sudo service heimdalld start Issue: Validator Heimdall is unable to connect to peers This typically means that your sentry Heimdall is running into issues. Solution - Check your sentry Heimdall to ensure that the service is running properly. - If the service is stopped, restarting it on your sentry node should resolve the issue. - Likewise, after addressing any issues with your sentry, restarting your Heimdall service should also help resolve the problem. # reporting-issues.md: Where to report a bug Any discovered bugs or vulnerabilities related to our Bug Bounty Program should be reported as follows: - For websites and applications: https://hackerone.com/polygon-technology - For smart contracts: https://immunefi.com/bounty/polygon - For security inquiries, please contact us at security@polygon.technology. (Please disclose vulnerabilities through the bug bounty program) !!! info Performing an attack and not providing submission of your proof will result in disqualification of your attempt. Make sure you add all relevant details such as your email address and Discord ID. Providing ample details creates a rapport of communication, and helps the Polygon team evaluate your submission appropriately. What happens after submitting a report Once an issue is reported, the Polygon team reviews it, comments, and updates on the status of the issue. After evaluation, the Polygon team reports the outcome of the submission. The severity of the issue also gets tagged as per the evaluation. Contact us for all other questions Via E-mail - For node operators: node-support@polygon.technology - For validators: validator-support@polygon.technology Via support portal - Visit the support portal atÂ support.polygon.technology. - Sign in using your email address and select Submit a Ticket located in the top right corner. !Polygon support portal - Tha above page appears. To ensure an accurate response, please include the following details when submitting your ticket: - The versions of Bor and Heimdall you are using. - At least one hour of logs related to the affected services. - TheÂ config.tomlÂ files for the affected services. Via Discord - Please visit theÂ #pos-full-node-queriesÂ channel on our Discord server and feel free to post your questions there. We will address your queries directly in that channel. # technical-faqs.md: 1. Are the private keys same for Heimdall and Bor keystore? Yes, the private key used for generating Validator keys and Bor Keystore is the same. The private key used in this instance is your Wallet's ETH address where your Polygon testnet tokens are stored. 2. List of Common Commands Refer to the list of common commands that might come in handy while troubleshooting. 3. Default Directories - Heimdall genesis file: /var/lib/heimdall/config/genesis.json - Heimdall-config.toml file: /var/lib/heimdall/config/heimdall-config.toml - Heimdall config.toml file: /var/lib/heimdall/config/config.toml - Heimdall data directory: /var/lib/heimdall/data/ - Bor config.toml file: /var/lib/bor/config.toml - Bor data directory: /var/lib/bor/data/bor/chaindata 4. From where do I create the API key? You can access this link: https://infura.io/register . Make sure that once you have setup your account and project, you copy the API key for Sepolia and not mainnet. Mainnet is selected by default. 5. How do I delete remnants of Heimdall and Bor? You can follow the following commands to delete the remnants of Heimdall and Bor from your machines. For the Linux package, run: $ sudo dpkg -i bor And delete the Bor directory using: $ sudo rm -rf /var/lib/bor For binaries, run: $ sudo rm -rf /var/lib/bor And then run: $ sudo rm /var/lib/heimdall 6. How many validators can be active concurrently? Under the current limit, a maximum of 105 validators can be active at any given time. It's important to note that active validators are primarily those with high uptime, while participants with significant downtime may be removed. 7. How much should I stake? A minimum stake of 10,000 POL tokens is required (as per PIP-4). We recommend setting a Heimdall fee of 10 POL. 8. I'm not clear on which Private Key should I add when I generate validator key. The private key to be used is your wallet's ETH address where your Polygon testnet tokens are stored. You can complete the setup with one public-private key pair tied to the address submitted on the form. 9. Is there a way to know if Heimdall is synced? You can run the following command to check it: bash $ curl http://localhost:26657/status Check the value of the catchingup flag. If it is false then the node is all synced up. 10. Which file do I add the API key in? Once you have created the API key, you need to add it to the heimdall-config.toml file. 11. How to check if the correct signer address is used for validator setup? To check the signer address, run the following commandÂ on the validator node: bash heimdalld show-account 12. Error: Failed to unlock account (0x...) No key for given address or file This error occurs because the path for the password.txt file is incorrect. You can follow the below steps to rectify this: 1. Copy the Bor keystore file to /var/lib/bor/keystore 2. Copy password.txt to /var/lib/bor/ 3. Make sure you have added correct address in /var/lib/bor/config.toml. 4. Ensure thatÂ the privvalidatorkey.jsonÂ andÂ UTC--

Â files have relevant permissions. To set relevant permissions forÂ privvalidatorkey.json, runÂ sudo chown -R heimdall:nogroup /var/lib/heimdall/config/privvalidatorkey.json, and similarly, runÂ sudo chown -R bor:nogroup /var/lib/bor/data/keystore/UTC--

Â forÂ the UTC--

file. 13. My node is not signing any checkpoints Try the following solutions: 1. Start by checking and updating the borrpcurl parameter in the heimdall-config.toml file of the validator to any external node RPC providers and restart the services. This change helps to avoid missing checkpoints. !!! info At this point in time, the node will not mine blocks. So once the issue is fixed, the changes made have to be reverted for the node to return to normal functionality. 2. Verify that the Heimdall service is running normally on both your sentry and validator nodes. If the service has stopped unexpectedly or is encountering errors, attempt to restart it and check if it resumes normal operation. 3. Check your Bor service logs for any errors or signs of abrupt halting. Try restarting your Bor service to resolve the issue. 4. If these steps don't resolve the issue, please contact our support team and share the relevant logs for further assistance. 14. Consequences of a validator missing checkpoints - Economic impact - Loss of reputation as a reliable validator - Missed node rewards for delegators - Repeatedly missing checkpoints can lead to grace period one and two, followed by final notice and removal from the network. # validator-ansible.md: This section guides you through starting and running the validator node through an Ansible playbook. Check out the Git repository for details. !!! info "Limited spots for new validators" There is limited space for accepting new validators. New validators can only join the active set when an already active validator unbonds. Prerequisites Three machines: One local machine on which you will run the Ansible playbook; two remote machines â€" one sentry and one validator. Ansible installed on the local machine. Python 3.x installed on the local machine. On the remote machines, make sure Go is not installed. Your local machine's SSH public key added to the remote machines, allowing Ansible to connect to them. Overview To deploy a running validator node, follow these steps in the exact sequence: !!! warning Performing these steps out of sequence may lead to configuration issues. It's crucial to note that setting up a sentry node must always precede the configuration of the validator node. 1. Have the three machines prepared. 2. Set up a sentry node through Ansible. 3. Set up a validator node through Ansible. 4. Configure the sentry node. 5. Start the sentry node. 6. Configure the validator node. 7. Set the owner and signer keys. 8. Start the validator node. Set up the sentry node On your local machine, clone the node-ansible repository: sh git clone https://github.com/maticnetwork/node-ansible Change the working directory to the cloned repository using: sh cd node-ansible Add the IP addresses of the remote machines that will become a sentry node and a validator node to the inventory.yml file. yml all: hosts: children: sentry: hosts: xxx.xxx.xx.xx: <----- Add IP for sentry node xxx.xxx.xx.xx: <----- Add IP for second sentry node (optional) validator: hosts: xxx.xx.xx.xx: <----- Add IP for validator node Example: yml all: hosts: children: sentry: hosts: 188.166.216.25: validator: hosts: 134.209.100.175: Check that the remote sentry machine is reachable. On the local machine, run: sh ansible sentry -m ping You should get this as output: sh xxx.xxx.xx.xx | SUCCESS => { "ansiblefacts": { "discoveredinterpreterpython": "/usr/bin/python3" }, "changed": false, "ping": "pong" } Do a test run of the sentry node setup by running the following command: sh ansible-playbook playbooks/network.yml --extra-var="borversion=v1.3.7 heimdallversion=v1.0.7 network=mainnet nodetype=sentry" --list-hosts You should see an output like this: sh playbook: playbooks/network.yml pattern: ['all'] host (1): xx.xxx.x.xxx Run the sentry node setup with sudo privileges: sh ansible-playbook -l sentry playbooks/network.yml --extra-var="borversion=v1.1.0 ansible-playbook playbooks/network.yml --extra-var="borversion=v1.3.7 heimdallversion=v1.0.7 network=mainnet nodetype=sentry" Once the setup is complete, you will see a message of completion on the terminal. !!! tip "How to start over" If you run into an issue and would like to start over, run: sh ansible-playbook -l sentry playbooks/clean.yml Set up the validator node At this point, you have the sentry node set up. Your local machine is also configured with an Ansible playbook to run the validator node setup. To check that the remote validator machine is reachable, run ansible validator -m ping on your local machine. You should get the following output: sh xxx.xxx.xx.xx | SUCCESS => { "ansiblefacts": { "discoveredinterpreterpython": "/usr/bin/python3" }, "changed": false, "ping": "pong" } Do a test run of the validator node setup by running the following command: sh ansible-playbook -l validator playbooks/network.yml --extra-var="borversion=v1.1.0 ansible-playbook playbooks/network.yml --extra-var="borversion=v1.3.7 heimdallversion=v1.0.7 networkversion=mainnet-v1 nodetype=validator

heimdallnetwork=mainnet" --list-hosts You should see an output like this: sh playbook: playbooks/network.yml pattern: ['all'] host (1): xx.xxx.x.xxx Run the validator node setup with sudo privileges: sh ansible-playbook playbooks/network.yml --extra-var="borversion=v1.3.7 heimdallversion=v1.0.7 networkversion=mainnet-v1 nodetype=validator heimdallnetwork=mainnet" Once the setup is complete, you will see a message of completion on the terminal. !!! info If you run into an issue and would like to start over, run: sh ansible-playbook -l validator playbooks/clean.yml Configure the sentry node Start by logging into the remote sentry machine. Configure Heimdall Open the config file for editing by running vi /var/lib/heimdall/config/config.toml. In the config.toml file, update the following parameters: moniker â€" any name. Example: moniker = "my-full-node". seeds â€" the seed node addresses consisting of a node ID, an IP address, and a port. Use the following values for mainnet: toml seeds="1500161dd491b67fb1ac81868952be49e2509c9f@52.78.36.216:26656,dd4a3f1750af5765266231b9d8ac764599921736@3.36.224.80:26656,8ea4f592ad6cc38d7532aff418d1fb97052463af@34.240.245.39:26656,e !!! info "Amoy node seeds" The Heimdall and Bor seeds don't need to be configured manually for Amoy testnet since they've already been included at genesis. pex â€" set the value to true to enable the peer exchange. Example: pex = true. privatepeerids â€" the node ID of Heimdall set up on the validator machine. To get the node ID of Heimdall on the validator machine: 1. Log into the validator machine. 2. Run: heimdalld tendermint show-node-id. Example: privatepeerids = "0ee1de0515f577700a6a4b6ad882eff1eb15f066". prometheus â€" set the value to true to enable the Prometheus metrics. Example: prometheus = true. maxopenconnections â€" set the value to 100. Example: maxopenconnections = 100. Finally, save the changes in config.toml. Configure the Bor service Open the config file for editing by running: vi /var/lib/bor/config.toml. In config.toml, ensure the boot node addresses consisting of a node ID, an IP address, and a port by adding them under bootnodes in [p2p.discovery] section: toml -bootnodes "enode://b8f1cc9c5d4403703fbf377116469667d2b1823c0daf16b7250aa576bacf399e42c3930ccfcb02c5df6879565a2b8931335565f0e8d3f8e72385ecf4a4bf160a@3.36.224.80:30303", "enode://8729e0c825f3d9cad382555f3e46dcff21af323e89025a0e6312df541f4a9e73abfa562d64906f5e59c51fe6f0501b3e61b07979606c56329c020ed739910759@54.194.245.5:30303" !!! info "Amoy node seeds" The Heimdall and Bor seeds don't need to be configured manually for Amoy testnet since they've already been included at genesis. In config.toml, ensure the static-nodes parameter has the following values: "enode://validatormachineenodeID@validatormachineip:30303" â€" the node ID and IP address of Bor set up on the validator machine. To get the node ID of Bor on the validator machine: 1. Log into the validator machine. 2. Run:Â bor attach /var/lib/bor/bor.ipc 3. Run: admin.nodeInfo.enode !!! info Please note that the IPC console is only accessible when Bor is running. To get the enode of the validator node, setup the validator node and then run the above commands. Finally, save the changes in config.toml. Configure firewall The sentry machine must have the following ports accessible from the public internet 0.0.0.0/0: Port 26656- Your Heimdall service will connect your node to other nodes' Heimdall service. Port 30303- Your Bor service will connect your node to other nodes' Bor service. !!! note "Sentry node with a VPN enabled" If the sentry node utilizes a VPN connection, it may restrict incoming SSH connections solely to the VPN IP address. Start the sentry node First, start the Heimdall service. Once the Heimdall service is fully synced, start the Bor service. !!! note "Syncing node using snapshots" The Heimdall service takes several days to fully sync from scratch. Alternatively, you can use a maintained snapshot, which will reduce the sync time to a few hours. For detailed instructions, see Snapshot Instructions for Heimdall and Bor. Start the Heimdall service Start the Heimdall service by running the following command: sh sudo service heimdalld start To check the Heimdall service logs, run the following command: sh journalctl -u heimdalld.service -f !!! bug "Common errors" In the logs, you may see the following errors: Stopping peer for error MConnection flush failed use of closed network connection These logs mean that one of the nodes on the network refused a connection to your node. Wait for your node to crawl more nodes on the network. You don't need to do anything manually to address these errors. Check the sync status of Heimdall using the following command: sh curl localhost:26657/status In the output, the catchingup value signifies the following: true â€" the Heimdall service is syncing. false â€" the Heimdall service is fully synced. Wait for the Heimdall service to fully sync. Start the Bor service Once the Heimdall service is fully synced, start the Bor service using the following command. sh sudo service bor start Check the Bor service logs using the following command: sh journalctl -u bor.service -f Configure the validator node !!! note "RPC endpoint" To complete this section, you must have your own RPC endpoint of your own fully synced Ethereum mainnet node ready. Configure the Heimdall service Log into the remote validator machine. Open the config file for editing by running: vi /var/lib/heimdall/config/config.toml. Next, update the following parameters in the config file: moniker â€" any name. Example: moniker = "my-validator-node". pex â€" set the value to false to disable the peer exchange. Example: pex = false. privatepeerids â€" comment out the value to disable it. Example: privatepeerids = "". To get the node ID of Heimdall on the sentry machine, run the following command: 1. Login to the sentry machine. 2. Run heimdalld tendermint show-node-id. Example: persistentpeers = "sentrymachineNodeID@sentryinstanceip:26656" prometheus â€" set the value to true to enable the Prometheus metrics. Example: prometheus = true. Save the changes in config.toml. Now, open heimdall-config.toml for editing by running: vi /var/lib/heimdall/config/heimdall-config.toml. In the heimdall-config.toml file, update the following parameters: ethrpcurl â€" an RPC endpoint for a fully synced Ethereum mainnet node, i.e Infura. ethrpcurl = Example: ethrpcurl = "https://nd-123-456-789.p2pify.com/60f2a23810ba11c827d3da642802412a" Finally, save the changes in heimdall-config.toml. Configure the Bor service Open the config file for editing by running: vi /var/lib/bor/config.toml. Update the value of static-nodes parameter as follows: toml static-nodes = [""] To get the node ID of Bor on the sentry machine: 1. Log into the sentry machine. 2. RunÂ bor attach /var/lib/bor/bor.ipc 3. Run admin.nodeInfo.enode Finally, save the changes in config.toml file. Set the owner and signer key On Polygon PoS, it is recommended that you keep the owner and signer keys different. Signer: The address that signs the checkpoint transactions. It is advisable to keep at least 1 ETH on the signer address. Owner: The address that does the staking transactions. It is advisable to keep the POL tokens on the owner address. Generate a Heimdall private key !!! warning The Heimdall private key must be generated only on the validator machine. Do not generate it on the sentry machine. To generate the private key, run: sh heimdallcli generate-validatorkey ETHEREUMPRIVATEKEY Here ETHEREUMPRIVATEKEY is your Ethereum walletâ€™s signer private key. This will generate the privvalidatorkey.json file. Move the newly generated JSON file to the Heimdall configuration directory using the following command: sh mv ./privvalidatorkey.json /var/lib/heimdall/config/ Generate a Bor keystore file !!! warning The Bor keystore file must be generated only on the validator machine. Do not generate it on the sentry machine. To generate the private key, run: sh heimdallcli generate-keystore ETHEREUMPRIVATEKEY Here ETHEREUMPRIVATEKEY is your Ethereum walletâ€™s signer private key. When prompted, set up a password to the keystore file. This will generate a UTC--
keystore file. Move the generated keystore file to the Bor configuration directory: sh mv ./UTC--
/var/lib/bor/data/keystore/ Add password.txt Make sure to create a password.txt file, and then add the Bor keystore file password right in the /var/lib/bor/password.txt file. Add your Ethereum address OpenÂ the config fileÂ for editing by running:Â vi /var/lib/bor/config.toml. bash [miner] mine = true etherbase = "validator address" [accounts] unlock = ["validator address"] password = "The path of the file you entered in password.txt" allow-insecure-unlock = true !!! warning Please ensure that privvalidatorkey.jsonÂ &Â UTC--
Â files have relevant permissions. To set relevant permissions forÂ privvalidatorkey.json, run the following command: sudo chown -R heimdall:nogroup /var/lib/heimdall/config/privvalidatorkey Similarly, to set the permissions for UTC--
, run the following command: sudo chown -R bor:nogroup /var/lib/bor/data/keystore/UTC--
Finally, save the changes in /var/lib/bor/config.toml. Start the validator node Start the Heimdall service First, start the Heimdall service on the validator machine. Once the Heimdall service is fully synced, you can run the Bor service. Start the Heimdall service using the following command: sh sudo service heimdalld start !!! info TheÂ heimdall-restÂ service and theÂ heimdall-bridge both start along with Heimdall. Check the Heimdall service logs by running the following command: sh journalctl -u heimdalld.service -f Check the sync status of Heimdall by running the following command: sh curl localhost:26657/status In the output, the catchingup value signifies the following: true â€" the Heimdall service is syncing. false â€" the Heimdall service is fully synced. Wait for the Heimdall service to fully sync. Start the Bor service Once the Heimdall service on the validator machine is fully synced, start the Bor service on the validator machine. Start the Bor service by running the following command: sh sudo service bor start Check the Bor service logs by running the following command: sh journalctl -u bor.service -f # validator-best-practices.md: This document explores best practices for running a Polygon PoS validator node. !!! info "Deploying a validator node" Get started with the process of deploying a validator node by reading the doc on node deployment prerequisites. Owner and signer wallets The signer wallet is an address that is used for signing Heimdall blocks, checkpoints, and other signing-related activities. This wallet's private key is on the validator node for signing purposes. It cannot manage staking, rewards, or delegations. The validator must keep two types of balances in this wallet: - POL tokens on Heimdall (through top-up transactions) to perform validator responsibilities on Heimdall. - ETH on Ethereum to send checkpoints on Ethereum. The owner wallet is an address that is used for staking, re-staking, changing the signer key, withdrawing rewards, and managing delegation related parameters on the Ethereum chain. !!! warning The private key for this address MUST be kept secure. All transactions through this key happen on the Ethereum chain. !!! tip Validators are advised to take all precautions to safely generate and store wallet keys. It is important to not expose key details to anyone that does not require the information. Wallet setup and maintenance - Key storage and maintenance are critically important. Use a secrets manager, or password manager for key management when setting up a wallet for your validator. If you choose to use another method, please ensure you have a solid understanding of the associated processes. - Refer to OWASP's key management cheat sheet. - Hardware wallets can provide an additional layer of security, but do not rely on them to be completely fault proof, or to protect you in case your key is compromised. - Rotate keys at least once a year. Also, it is important to rotate your keys if, at any point, you believe them to be compromised. - Please do your own research for your specific needs. Operating system The operating system of your choice is also a key factor in securing your validator node and preventing exploitation. - Please keep your system up to date using common practices like using your operating system package manager, such as yum, apt, brew, pacman, etc. There are many package managers, and each one has its own best practices guide and suggestions that you can refer to. - NEVER run unnecessary software on your node. Limit the number of services and applications running on your node. - Run single-purpose nodes. For example, you should not run a validator as an RPC endpoint for the public to consume. This is a known issue which you can avoid by not using the node for multiple purposes. - Validator nodes should only be running the required Bor, Heimdall, and RabbitMQ services. Anything else running on the host, outside of a monitoring tool or firewall, could open you up to exploits. - Limit access to your validator node to as few people as possible. This includes limiting SSH access only to selected people, or having no access to the host at all. The fewer people with access to the nodes, the fewer potential key disclosures, accidental events, and attack surfaces. - Accessing your machine can be accomplished in a couple of ways. On AWS, you could use an SSM or GCP cloud shell. You can also use Docker containers if you wish to rely on SSH. Please understand the risks of key and credential disclosure and any accounts that have access to the host(s). If you require SSH access, and need to expose the port publicly, the following best practices can increase security: - Use IP-restricted SSH access on a different port. - Use SSH keys and not passwords. - Disable password login. - Disable all root access; use individual accounts with sudo. - Add brute-force SSH protection: Fail2Ban. - Add 2FA for SSH access. Networking - Do not expose any ports publicly for your validator node. Instead, use a sentry/full node configured with port 30303 for Bor and port 26656 for Heimdall available publicly. - Configure your validator to operate as a static node in the Bor config.toml file and as a seed node/persistent peer in the Heimdall config. This allows your validator node to be isolated from the public internet and uses your sentry/full node instead of relying on unknown peers, which allows for greater security. An example of this looks like the following: !network setup In this example, the validator sends outbound requests to the network but does not allow any inbound connections, except for the sentry/full node we already configured. Note that this does not mean you can be careless with your network configurations and operating system maintenance for the Sentry node. It is just a way to avoid and mitigate against common attack vectors. There are a number of ways to achieve this. If you are using AWS, you can set up your security groups to only allow traffic from the sentry node to your validator, and vice versa. In this scenario, you must make sure that you have proper monitoring and observability tools for your network. !!! warning "Docker and UFW" Be careful when using UFW to restrict ports if your services are using Docker, as Docker automatically opens the Linux firewall for ports it "maps" to the host, therefore bypassing UFW rules. For more information, check this post addressing the Docker published ports ignoring firewall rules. Every cloud provider offers a number of tools for this type of networking, including VPC, Security Group, or equivalent. This method, while more secure than leaving the hosts exposed on the Internet, has overheads of engineering time, resource management, and observability. !!! warning "Public ports" A validator node, under no circumstances, should have any ports reachable by the public internet. In your topology, you may have other scenarios to consider. Beware of the risks involved with what ports or nodes have access to your host(s). Validator backup There is also now a validator backup playbook available from the node-ansible repo. This tool allows you to backup your current configuration for your validator. This can be quite useful for migrating to a new host. To use this tool, run the following command: bash ansible-playbook -i $inventory playbooks/validator-backup.yml -e "destination=$WHEREYOUWANTTOSAVELOCALLY borpath=PATHTOYOURBORINSTALL heimdallpath=PATHTOYOURHEIMDALLPATH" This tool requires you to define a destination variable, the path to your Bor config location, and the path to your Heimdall config location. It then creates a tarball and stores it locally on your machine in the directory path you have provided. Host migration If you are migrating a host, you may want to create an existing snapshot of your chain data for Bor and Heimdall. You can do this by running the following commands: bash ansible-playbook -i $inventory playbooks/bor/snapshot-create -e "chaindata=$path target=$targetsavedir" ansible-playbook -i $inventory playbooks/heimdall/snapshot-create -e "data=$path target=$targetsavedir" This starts a screen session while generating the tarball, which also implies that you have enough disk space on the host that you defined as your target variable to store the output. From here, you can copy the tarball to any host. Monitoring and observability If you already ship your logs to Coralogix, Datadog, or Splunk, then your monitoring uses this information. You could also be using Nagios or another similar tool to actively monitor your validator nodes. Here are some scenarios and data to consider logging when monitoring a node: - Is the Bor service up? - Is the Heimdall service up? - Is the chain out of sync? - Traffic inbound (RPC calls to your sentry). - Disk space usage. - Memory usage. - Network usage. How to best monitor these depends on your logging set up and requirements. Monitoring is a cost that may ultimately provide you peace of mind as a node operator. Make sure to balance the signal-to-noise ratio for the alerts. This usually takes some fine-tuning. # validator-binaries.md: This guide will walk you through running a Polygon validator node using binaries. !!! info "Limited spots for new validators" There is limited space for accepting new validators. New validators can only join the active set when an already active validator unbonds. Prerequisites Two machines - one sentry and one validator. build-essential installed on both the sentry and the validator machines. To install: bash sudo apt-get install build-essential Go 1.19 installed on both the sentry and the validator machines. To install: bash wget https://raw.githubusercontent.com/maticnetwork/node-ansible/master/go-install.sh bash go-install.sh sudo ln -nfs /.go/bin/go /usr/bin/go RabbitMQ installed on both the sentry and the validator machines. Here are the commands to install RabbitMQ: bash sudo apt-get update sudo apt install build-essential sudo apt install erlang wget https://github.com/rabbitmq/rabbitmq-server/releases/download/v3.10.8/rabbitmq-server3.10.8-1all.deb sudo dpkg -i rabbitmq-server3.10.8-1all.deb Overview To set up a running validator node, follow these steps in the exact sequence: !!! warning Performing these steps out of sequence may lead to configuration issues. It's crucial to note that setting up a sentry node must always precede the configuration of the validator node. 1. Prepare two machines, one for the sentry node and one for the validator node. 2. Install the Heimdall and Bor binaries on the sentry and validator machines. 3. Set up the Heimdall and Bor service files on the sentry and validator machines. 4. Configure the sentry node. 5. Start the sentry node. 6. Configure the validator node. 7. Set the owner and signer keys. 8. Start the validator node. Installing the binaries Polygon node consists of 2 layers: Heimdall and Bor. Heimdall is a tendermint fork that monitors contracts in parallel with the Ethereum network. Bor is basically a Geth fork that generates blocks shuffled by Heimdall nodes. Both binaries must be installed and run in the correct order to function properly. Install Heimdall Install the latest version of Heimdall and related services. Make sure you checkout to the correct release version. To install Heimdall, run the following commands: bash curl -L https://raw.githubusercontent.com/maticnetwork/install/main/heimdall.sh | bash -s -- You can run the above command with following options: - heimdallversion: valid v1.0+ release tag from https://github.com/maticnetwork/heimdall/releases - networktype: mainnet and amoy - nodetype: sentry That will install the heimdalld and heimdallcli binaries. Verify the installation by checking the Heimdall version on your machine using the following command: bash heimdalld version --long !!! note Before proceeding, ensure that Heimdall is installed on both the sentry and validator machines. Install Bor Install the latest version of Bor. Make sure you checkout to the correct release version. bash curl -L https://raw.githubusercontent.com/maticnetwork/install/main/bor.sh | bash -s -- You can run the above command with following options: - borversion: valid v1.0+ release tag from https://github.com/maticnetwork/bor/releases - networktype: mainnet and amoy - nodetype: sentry That will install the bor binary. Verify the installation by checking the Bor version on your machine: bash bor version !!! note Before proceeding, make sure that Bor is installed on both the sentry and validator machines. Configuring the sentry node Start by logging in to the remote sentry machine. Configure Heimdall Open the Heimdall configuration file for editing: sh vi /var/lib/heimdall/config/config.toml In config.toml, change the following parameters: moniker â€" any name. Example: moniker = "my-sentry-node". seeds â€" the seed node addresses consisting of a node ID, an IP address, and a port. Use the following values for mainnet: toml seeds="1500161dd491b67fb1ac81868952be49e2509c9f@52.78.36.216:26656,dd4a3f1750af5765266231b9d8ac764599921736@3.36.224.80:26656,8ea4f592ad6cc38d7532aff418d1fb97052463af@34.240.245.39:26656,e !!! info "Amoy node seeds" The Heimdall and Bor seeds don't need to be configured manually for Amoy testnet since they've already been included at genesis. pex â€" set the value to true to enable the peer exchange. Example: pex = true. privatepeerids â€" the node ID of Heimdall set up on the validator machine. To get the node ID of Heimdall on the validator machine: 1. Log in to the validator machine. 2. Run: sh heimdalld tendermint show-node-id Example: privatepeerids = "0ee1de0515f577700a6a4b6ad882eff1eb15f066". prometheus â€" set the value to true to enable the Prometheus metrics. Example: prometheus = true. maxopenconnections â€" set the value to 100. Example: maxopenconnections = 100. Finally, save the changes in config.toml. Configure Bor Open the Bor configuration file for editing using: sh vi /var/lib/bor/config.toml In config.toml, add the boot node addresses consisting of a node ID, an IP address, and a port by adding them under bootnodes in [p2p.discovery] section: config --bootnodes "enode://b8f1cc9c5d4403703fbf377116469667d2b1823c0daf16b7250aa576bacf399e42c3930ccfcb02c5df6879565a2b8931335565f0e8d3f8e72385ecf4a4bf160a@3.36.224.80:30303", "enode://8729e0c825f3d9cad382555f3e46dcff21af323e89025a0e6312df541f4a9e73abfa562d64906f5e59c51fe6f0501b3e61b07979606c56329c020ed739910759@54.194.245.5:30303" !!! info "Amoy node seeds" The Heimdall and Bor seeds don't need to be configured manually for Amoy testnet since they've already been included at genesis. Now, add the following in the config.toml file: toml [p2p] [p2p.discovery] static-nodes = [""] To fetch the Node ID of Bor on the validator machine: - Log in to the validator machine - RunÂ bor attach /var/lib/bor/bor.ipc - Run admin.nodeInfo.enode !!! info PC console is only accessible when Bor is running. To get the enode of the validator node, setup the validator node and then run the above commands. Finally, save the changes in config.toml. Configuring a firewall The sentry machine must have the following ports open to the world

0.0.0.0/0: 26656- Your Heimdall service will connect your node to other nodes Heimdall service. 30303- Your Bor service will connect your node to other nodes Bor service. Starting the sentry node First, start the Heimdall service. Then, once the Heimdall service syncs, start the Bor service. !!! info "Sync node using snapshots" The Heimdall service can take several days to sync from scratch fully. Alternatively, you can use a maintained snapshot, which will reduce the sync time to a few hours. For detailed instructions, see Snapshot Instructions for Heimdall and Bor. Starting the Heimdall service Start the Heimdall service: sh sudo service heimdalld start !!! info The heimdall-rest service starts along with heimdall. Check the Heimdall service logs using the following command: sh journalctl -u heimdalld.service -f !!! bug "Common error" In the logs, you may see the following errors: Stopping peer for error MConnection flush failed use of closed network connection These logs mean that one of the nodes on the network refused a connection to your node. Wait for your node to crawl more nodes on the network; you do not need to do anything manually to address these errors. Check the sync status of Heimdall using the following command: sh curl localhost:26657/status In the output, the catchingup signifies the following: true: The Heimdall service is syncing. false: The Heimdall service is fully synced. Wait for the Heimdall service to sync fully. Starting the Bor service Once the Heimdall service is fully synced, start the Bor service. Start the Bor service: sh sudo service bor start Check the Bor service logs: sh journalctl -u bor.service -f Configuring the validator node !!! info In order to proceed, you'll need to have access to an RPC endpoint of a fully synced Ethereum mainnet node ready. Configuring the Heimdall service Log in to the remote validator machine. Open for editing vi /var/lib/heimdall/config/config.toml. In config.toml, change the following: moniker â€" any name. Example: moniker = "my-validator-node". pex â€" set the value to false to disable the peer exchange. Example: pex = false. privatepeerids â€" comment out the value to disable it. Example: privatepeerids = "". To get the node ID of Heimdall on the sentry machine: 1. Log in to the sentry machine. 2. Run heimdalld tendermint show-node-id. Example: persistentpeers = "sentrymachineNodeID@sentryinstanceip:26656" prometheus â€" set the value to true to enable the Prometheus metrics. Example: prometheus = true. Save the changes in config.toml. Open for editing vi /var/lib/heimdall/config/heimdall-config.toml. In heimdall-config.toml, update the following: ethrpcurl â€" an RPC endpoint for a fully synced Ethereum mainnet node, e.g., Infura. ethrpcurl = Example: ethrpcurl = "https://nd-123-456-789.p2pify.com/60f2a23810ba11c827d3da642802412a" Save the changes in heimdall-config.toml. Configuring the Bor service Open config file for editing using: vi /var/lib/bor/config.toml Change the value of static-nodes parameter as follows: json static-nodes = [""] // the node ID and IP address of Bor set up on the sentry machine To get the Node ID of Bor on the sentry machine: 1. Log in to the sentry machine. 2. Run bor attach /var/lib/bor/bor.ipc. 3. Run admin.nodeInfo.enode. Setting the Owner and Signer Key On Polygon PoS, it is recommended that you keep the owner and signer keys different. Signer: The address that signs the checkpoint transactions. It is advisable to keep at least 1 ETH on the signer address. Owner: The address that is used to perform the staking transactions. It is advisable to keep the POL tokens on the owner address. Generating a Heimdall private key You must generate a Heimdall private key only on the validator machine. Do not generate a Heimdall private key on the sentry machine. To generate the private key, run: sh heimdallcli generate-validatorkey ETHEREUMPRIVATEKEY where ETHEREUMPRIVATEKEY is your Ethereum walletâ€™s private key. This will generate privvalidatorkey.json. Move the generated JSON file to the Heimdall configuration directory using the following command: sh mv ./privvalidatorkey.json /var/lib/heimdall/config Generating a Bor keystore file You must generate a Bor keystore file only on the validator machine. Do not generate a Bor keystore file on the sentry machine. To generate the private key, run: sh heimdallcli generate-keystore ETHEREUMPRIVATEKEY where ETHEREUMPRIVATEKEY is your Ethereum walletâ€™s private key. When prompted, set up a password to the keystore file. This will generate a UTC-- keystore file. Move the generated keystore file to the Bor configuration directory using the following command: sh mv ./UTC--
/var/lib/bor/data/keystore Add password.txt Make sure to create a password.txt file, then add the Bor keystore file password in the /var/lib/bor/password.txt file. Add your Ethereum address Open config.toml for editing: vi /var/lib/bor/config.toml. toml [miner] mine = true etherbase = "validator address" [accounts] unlock = ["validator address"] password = "The path of the file you entered in password.txt" allow-insecure-unlock = true !!! info "Set file permissions" Please ensure that privvalidatorkey.json & UTC--
files have relevant permissions. To set the permissions for privvalidatorkey.json, run: bash sudo chown -R heimdall:nogroup /var/lib/heimdall/config/privvalidatorkey.json and similarly, for the UTC--
file, run: bash sudo chown -R heimdall:nogroup /var/lib/bor/data/keystore/UTC--
Starting the Heimdall service You will now start the Heimdall service on the validator machine. Once the Heimdall service syncs, you will start the Bor service on the validator machine. Start the Heimdall service using the following command: sh sudo service heimdalld start !!! info The heimdall-rest service and heimdall-bridge starts along with heimdall. Check the Heimdall service logs using the following command: sh journalctl -u heimdalld.service -f Check the sync status of Heimdall using the following command: sh curl localhost:26657/status In the output, the catchingup signifies the following: true: The Heimdall service is syncing. false: The Heimdall service is synced. Wait for the Heimdall service to fully sync. Starting the Bor service Once the Heimdall service on the validator machine is fully synced, start the Bor service on the validator machine. Start the Bor service using the following command: sh sudo service bor start Check the Bor service logs using the following command: sh journalctl -u bor.service -f # validator-packages.md: This guide covers running a validator node through packages. Prerequisites Two machines â€" one sentry and one validator. Bash installed on both the sentry and the validator machines. RabbitMQ installed on both the sentry and the validator machines. See Downloading and Installing RabbitMQ. Overview To spin up a functioning validator node, follow these steps in the specified sequence: !!! warning Performing these steps out of sequence may lead to configuration issues. It's crucial to note that setting up a sentry node must always precede the configuration of the validator node. 1. Prepare two machines, one for the sentry node and one for the validator node. 2. Install the Heimdall and Bor binaries on the sentry and validator machines. 3. Set up the Heimdall and Bor services on the sentry and validator machines. 4. Configure the sentry node. 5. Start the sentry node. 6. Configure the validator node. 7. Set the owner and signer keys. 8. Start the validator node. Installing packages Heimdall Install the default latest version of sentry and validator for the PoS mainnet/Amoy testnet: shell curl -L https://raw.githubusercontent.com/maticnetwork/install/main/heimdall.sh | bash -s -- or install a specific version, node type (sentry or validator), and network (mainnet or amoy). All release versions can be found on Heimdall GitHub repository. shell Example: curl -L https://raw.githubusercontent.com/maticnetwork/install/main/heimdall.sh | bash -s -- v1.0.7 mainnet sentry Bor Install the default latest version of sentry and validator for the PoS mainnet/Amoy testnet: shell curl -L https://raw.githubusercontent.com/maticnetwork/install/main/bor.sh | bash or install a specific version, node type (sentry or validator), and network (mainnet or amoy). All release versions could be found on Bor Github repository. shell structure curl -L https://raw.githubusercontent.com/maticnetwork/install/main/bor.sh | bash -s -- Example: curl -L https://raw.githubusercontent.com/maticnetwork/install/main/bor.sh | bash -s -- v1.1.0 mainnet sentry Check installation Check Heimdall installation using the following command: shell heimdalld version --long Check Bor installation using the following command: shell bor version !!! info Before proceeding, please ensure Bor is installed on both the sentry and validator machines. Configure sentry node In this section, we will go through steps to initialize and customize configuration for sentry nodes. Configure Heimdall Open the Heimdall configuration file for editing using the following command: sh vi /var/lib/heimdall/config/config.toml In the config file, update the following parameters: moniker â€" any name. Example: moniker = "my-sentry-node". seeds â€" the seed node addresses consisting of a node ID, an IP address, and a port. Use the following values: toml seeds="1500161dd491b67fb1ac81868952be49e2509c9f@52.78.36.216:26656,dd4a3f1750af5765266231b9d8ac764599921736@3.36.224.80:26656,8ea4f592ad6cc38d7532aff418d1fb97052463af@34.240.245.39:26656,e pex â€" set the value to true to enable the peer exchange. Example: pex = true. privatepeerids â€" the node ID of Heimdall set up on the validator machine. To get the node ID of Heimdall on the validator machine: 1. Log in to the validator machine. 2. Run: sh heimdalld tendermint show-node-id Example: privatepeerids = "0ee1de0515f577700a6a4b6ad882eff1eb15f066". prometheus â€" set the value to true to enable the Prometheus metrics. Example: prometheus = true. maxopenconnections â€" set the value to 100. Example: maxopenconnections = 100. Finally, save the changes in config.toml. Configure Bor In /var/lib/bor/config.toml file, add the following: bash [p2p] [p2p.discovery] static-nodes = [""] To get the Node ID of Bor on the validator machine: 1. Log into the validator machine. 2. RunÂ bor attach /var/lib/bor/bor.ipc 3. Run admin.nodeInfo.enode !!! info The IPC console is only accessible when Bor is running. To get the enode of the validator node, setup the validator node, and then run the above commands. Example content of static node field in /var/lib/bor/config.toml: bash [p2p] [p2p.discovery] static-nodes = ["enode://410e359736bcd3a58181cf55d54d4e0bbd6db2939c5f548426be7d18b8fd755a0ceb730fe5cf7510c6fa6f0870e388277c5f4c717af66d53c440feedffb29b4b@134.209.100.175:30303"] Finally, save the changes in /var/lib/bor/config.toml. Configuring a firewall The sentry machine must have the following ports open to the public internet 0.0.0.0/0: Port 26656: Your Heimdall service will connect your node to other nodes Heimdall service. Port 30303: Your Bor service will connect your node to other nodes Bor service. Port 22: Open this port if your node is servicing validators. You will likely want to restrict what traffic can access this port as it is a sensitive port. Starting the sentry node First, start the Heimdall service. Once the Heimdall service is fully synced, you can start the Bor service. Reload service files Run the following command to reload service files to make sure all changes to service files are loaded correctly: sh sudo systemctl daemon-reload Starting the Heimdall service Start the Heimdall services using the following command: sh sudo service heimdalld start Check the Heimdall service logs using the following command: sh journalctl -u heimdalld.service -f !!! bug "Common errors" In the logs, you may see the following errors: Stopping peer for error MConnection flush failed use of closed network connection These logs mean that one of the nodes on the network refused a connection to your node. Wait for your node to crawl more nodes on the network; you do not need to do anything to address these errors. Check the Heimdalld logs using the following command: sh journalctl -u heimdalld.service -f Check the sync status of Heimdall using the following command: sh curl localhost:26657/status In the output, the catchingup value signifies the following: true: The Heimdall service is syncing. false: The Heimdall service is fully synced. Wait for the Heimdall service to sync fully. Starting the Bor service Once the Heimdall service is fully synced, start the Bor service using the following command: sh sudo service bor start Check the Bor service logs using the following command: sh journalctl -u bor.service -f Installing packages on the validator node Follow the same installation steps on the validator node. Configuring the validator node !!! note To complete this section, you must have an RPC endpoint of your fully synced Ethereum mainnet node ready. Configure Heimdall Log in to the remote validator machine. Open the Heimdall configuration file for editing using the following command: sh vi /var/lib/heimdall/config/config.toml In config.toml, update the following parameters: moniker â€" any name. Example: moniker = "my-validator-node". pex â€" set the value to false to disable the peer exchange. Example: pex = false. privatepeerids â€" comment out the value to disable it. Example: privatepeerids = "". To get the node ID of Heimdall on the sentry machine: 1. Log in to the sentry machine. 2. Run heimdalld tendermint show-node-id. Example: persistentpeers = "sentrymachineNodeID@sentryinstanceip:26656" prometheus â€" set the value to true to enable the Prometheus metrics. Example: prometheus = true. Save the changes in config.toml. Open the heimdall-config.toml file for editing by running: vi /var/lib/heimdall/config/heimdall-config.toml. In config file, update the following parameters: ethrpcurl â€" an RPC endpoint for a fully synced Ethereum mainnet node or testnet node, e.g., Infura. ethrpcurl = Example: ethrpcurl = "https://nd-123-456-789.p2pify.com/60f2a23810ba11c827d3da642802412a" Finally, save the changes in heimdall-config.toml. Configuring Bor In the /var/lib/bor/config.toml file, add the following: bash [p2p] [p2p.discovery] static-nodes = [""] To get the node ID of Bor on the sentry machine, run the following command: 1. Log into the sentry machine. 2. RunÂ bor attach /var/lib/bor/bor.ipc 3. Run admin.nodeInfo.enode !!! info The IPC console is only accessible when Bor is running. Example content of static-nodes field in /var/lib/bor/config.toml: bash [p2p] [p2p.discovery] static-nodes = ["enode://410e359736bcd3a58181cf55d54d4e0bbd6db2939c5f548426be7d18b8fd755a0ceb730fe5cf7510c6fa6f0870e388277c5f4c717af66d53c440feedffb29b4b@134.209.100.175:30303"] Finally, save the changes in /var/lib/bor/config.toml. Setting the Owner and Signer Key On Polygon PoS, it is recommended that you keep the owner and signer keys different. Signer: The address that signs the checkpoint transaction. It is advisable to keep at least 1 ETH on the signer address. Owner: The address that does the staking transactions. It is advisable to keep the POL tokens on the owner address. Generating a Heimdall private key You must generate a Heimdall private key only on the validator machine. Do not generate a Heimdall private key on the sentry machine. To generate the private key, run: sh heimdallcli generate-validatorkey ETHEREUMPRIVATEKEY where ETHEREUMPRIVATEKEY is your Ethereum wallet's private key. This will generate privvalidatorkey.json. Move the generated JSON file to the Heimdall configuration directory: sh mv ./privvalidatorkey.json /var/lib/heimdall/config Generating a Bor keystore file !!! warning You must generate a Bor keystore file only on the validator machine. Do not generate a Bor keystore file on the sentry machine. To generate the private key, run: sh heimdallcli generate-keystore ETHEREUMPRIVATEKEY where ETHEREUMPRIVATEKEY is your Ethereum wallet's private key. When prompted, set up a password to the keystore file. This will generate a UTC-- keystore file. Move the generated keystore file to the Bor configuration directory using the following command: sh mv ./UTC--
/var/lib/bor/data/keystore Add password.txt Make sure to create a password.txt file then add the Bor keystore file password right in the /var/lib/bor/password.txt file. Add your Ethereum address Open config file for editing by running: vi /var/lib/bor/config.toml. toml [miner] mine = true etherbase = "validator address" [accounts] unlock = ["validator address"] password = "The path of the file you entered in password.txt" allow-insecure-unlock = true !!! warning Please ensure that privvalidatorkey.json & UTC--
files have relevant permissions. To set relevant permissions for privvalidatorkey.json, run the following command: bash sudo chown -R heimdall:nogroup /var/lib/heimdall/config/privvalidatorkey.json And similarly, run the following command to set permissions for UTC--
: bash sudo chown -R heimdall:nogroup /var/lib/bor/data/keystore/UTC--
Starting the validator node At this point, you must have: The Heimdall service on the sentry machine syncs and is running. The Bor service on the sentry machine running. The Heimdall service and the Bor service on the validator machine configured. Your owner and signer keys configured. Reload service files Run the following command to reload the service files to make sure all changes to service files are loaded correctly: bash sudo systemctl daemon-reload Starting the Heimdall service Now, start the Heimdall service on the validator machine. Once the Heimdall service is fully synced, you can start the Bor service on the validator machine. !!! info In order to ensure you node is able to submit checkpoints normally, make sure that the -- bridge -- all flag is present correctly inÂ /lib/systemd/system/heimdalld.serviceÂ file. Start the Heimdall services by running the following command: sh sudo service heimdalld start Check the Heimdall service logs by running the following command: sh journalctl -u heimdalld.service -f Check the sync status of Heimdall by running the following command: sh curl localhost:26657/status In the output, the catchingup value signifies the following: true: The Heimdall service is syncing. false: The Heimdall service is synced. Wait for the Heimdall service to fully sync. Starting the Bor service Once the Heimdall service on the validator machine is fully synced, start the Bor service on the validator machine. Start the Bor service by running the following command: sh sudo service bor start Check the Bor service logs using the following command: sh journalctl -u bor.service -f # commands.md: This guide provides a curated list of common commands and Polygon PoS-specific operations essential for node operators. Whether you're setting up a full node, validator node or troubleshooting, these commands will assist you in managing your Polygon PoS environment effectively. Frequently used commands for Bor & Heimdall Bor To execute Bor IPC commands, use the following syntax: bash bor attach .bor/data/bor.ipc | IPC Command | RPC Command | Description | | --- | -- | --- | | admin.peers.length | curl -H "Content-Type: application/json" --data '{"jsonrpc": "2.0", "me "netpeerCount", "params": [], "id": 74}' localhost:8545 | Retrieves the number of peers connected to the node. | | admin.nodeInfo | | Provides detailed information about the node. | | eth.syncing | curl -H "Content-Type: application/json" -d '{"id":1, "jsonrpc":"2.0", "method": "ethsyncing","params": []}' localhost:8545 | Indicates whether the node is syncing (true) or not (false). | | eth.syncing.highestBlock - eth.syncing.currentBlock | | Compares the current block of your node to the highest block. | | eth.blockNumber | curl -H "Content-Type: application/json" -d '{"id":1, "jsonrpc":"2.0", "method": "ethblockNumber","params": []}' localhost:8545 | Returns the latest block number processed by the node. | | debug.setHead("0x"+((eth.getBlock('latest').number) - 1000).toString(16)) | | Rewinds the blockchain to 1000 blocks prior. | | admin.nodeInfo.enode | | Retrieves the public enode URL of the node. | | eth.syncing.currentBlock 100 / eth.syncing.highestBlock | | Calculates the remaining percentage for block synchronization. | | eth.getBlock("latest").number | curl http://YourIP:8545 -X POST -H "Content-Type: application/json" -d '{"jsonrpc":"2.0", "id":1, "method":"borgetSigners", "params":["0x98b3ea"]}' | Queries the height of the latest Bor block. | | | curl http://YourIP:8545 -X POST -H "Content-Type: application/json" --data '{"method":"ethchainId","params":[],"id":1,"jsonrpc":"2.0"}' | Retrieves the chainID. | Heimdall | Command | Description | | -- | -- | | curl localhost:26657/netinfo?jq .result.npeers | Returns the number of connected peers. | | curl -s localhost:26657/status | jq .result.syncinfo.latestblockheight | Retrieves Heimdall's current block height. | | curl localhost:26657/netinfo | grep moniker | Queries the node using its moniker. | | curl -s localhost:26657/status | jq .result.syncinfo.catchingup | Checks if Heimdall is in sync. | | curl -s localhost:26657/status | jq .result | jq .syncinfo | Verifies Heimdall's sync status. | | heimdalld unsafe-reset-all | Resets the database in case of issues. | | curl localhost:26657/status | Provides comprehensive information about Heimdall. | Node management commands | Description | Command | | --------------------------- | --- | | Locate Heimdall genesis file | $CONFIGPATH/heimdall/config/genesis.json | | Locate heimdall-config.toml | /etc/heimdall/config/heimdall-config.toml | | Locate config.toml | /etc/heimdall/config/config.toml | | Start Heimdall | $ sudo service heimdalld start | | Locate Bor genesis file | $CONFIGPATH/bor/genesis.json | | Start Bor | sudo service bor start | | Retrieve Heimdall logs | /var/log/matic-logs/ | | Check Heimdall logs | journalctl -fu heimdalld.service | | Check Bor logs | journalctl -fu bor.service | Remove Heimdall directories bash sudo rm -rf /var/lib/heimdalld/ Remove Bor directories bash sudo rm -rf /var/lib/bor # commit-chain-multisigs.md: Purpose and capabilities The primary role of multi-signature wallets (multisigs) is to facilitate contract upgrades during the early stages of development. As these contracts become more robust, Polygon plans to: - Transition from multisigs to governance-controlled proxies. - Implement timelocks for added security. - Phase out multisigs entirely in the long term. !!! info "Censorship" It's important to note that the existing multisigs do not have the capability to censor transactions, including bridge transactions. Active multi-signature wallets Ethereum chain multisigs | Multisig Address | $5/9$ Multisig
0xFa7D2a996aC6350f4b56C043112Da0366a59b74c | |:----------------:|:--:| | Purpose | To upgrade PoS and staking contracts on Ethereum. | | Chain | Ethereum | | Rights | - Update staking contracts for optimizations and upgrades.
- Address unexpected bugs in PoS contracts. | | Signatories | Quickswap, Curve, Polygon, Horizon Games, Cometh | Polygon commitchain multisigs | Multisig Address | $5/8$ Multisig
0x355b8E02e7F5301E6fac9b7cAc1D6D9c86C0343f | |:----------------:|:--:| | Purpose | To update custom ChildERC20s on Polygon commitchain. | | Chain | Polygon commitchain | | Rights | Ability to upgrade custom child contracts. | | Signatories | Quickswap, Curve, Polygon, Horizon Games, Cometh | Custom child ERC20s mapping | Multisig Address | $4/8$ Multisig
0x424bDE99FCfB68c5a1218fd3215caFfD031f19C4 | |:----------------:|:--:| | Purpose | To enable the mapping of custom ChildERC20s with Mainnet contracts. | | Chain | Ethereu

| | Rights | Limited to mapping; no access to Child tokens or deposit/withdrawal rights. | | Signatories | Polygon | Permissionless mapping | Multisig address | Permissionless mapping of standard ChildERC20 tokens (no multisig required) | |:-----------------|:---| | Purpose | FxPortal supports permissionless token mapping of standard ChildERC20 for any ERC20 token on Ethereum. | | Cha Permissionless | | Rights | Permissionless | | Signatories | Permissionless | Plans are underway to transition these functions to governance. We are currently exploring options such as Aave's governance contracts and Compoundâ€™s timelock contracts. # mapped-tokens.md: ï»¿This document provides examples of token mappings on the Polygon PoS mainnet and Amoy testnet. !!! tip "Token mapping" Want to get a custom token mapped to Polygon PoS so it can be bridged over? Send in your mapping request via this Google form. Amoy | Token name | Bridge | Parent chain address: Sepolia | Child chain: Amoy | | ---------------- | ------ | -- | -- | | PoS\-WETH | PoS | 0x700dDE29De87ed2c01c27C896dc8Badb4f671302 | 0x52eF3d68BaB452a294342DC3e5f464d7f610f72E | | DummyERC20Token | PoS | 0xb480378044d92C96D16589Eb95986df6a97F2cFB | 0xf3202E7270a10E599394d8A7dA2F4Fbd475e96bA | | DummyERC721Token | PoS | 0x421DbB7B5dFCb112D7a13944DeFB80b28eC5D22C | 0x02f83d4110D3595872481f677Ae323D50Aa09209 | | DummyERC1155Token | PoS | 0x095DD31b6473c4a32548d2A5B09e0f2F3F30d8F1 | 0x488AfDFef019f511E343becb98B7c24ee02fA639 | Mainnet | Token name | Bridge | Parent chain address: Ethereum mainnet | Child chain: Polygon mainnet | | ---------------- | ------ | -- | -- | | PoS\-WETH | PoS | 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE | 0x7ceB23fD6bC0adD59E62ac25578270cFf1b9f619 | | PoS\-USDC | PoS | 0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48 | 0x2791Bca1f2de4661ED88A30C99A7a9449Aa84174 | | PoS\-COMP | PoS | 0xc00e94cb662c3520282e6f5717214004a7f26888 | 0x8505b9d2254A7Ae468c0E9dd10Ccea3A837aef5c | | PoS\-LEND | PoS | 0x80fB784B7eD66730e8b1DBd9820aFD29931aab03 | 0x313d009888329C9d1cf4f75CA3f32566335bd604 | | PoS\-YFI | PoS | 0x0bc529c00C6401aEF6D220BE8C6Ea1667F6Ad93e | 0xDA537104D6A5edd53c6fBba9A898708E465260b6 | | PoS\-USDT | PoS | 0xdac17f958d2ee523a2206206994597c13d831ec7 | 0xc2132D05D31c914a87C6611C10748AEb04B58e8F | | PoS\-DAI | PoS | 0x6b175474e89094c44da98b954eedeac495271d0f | 0x8f3Cf7ad23Cd3CaDbD9735AFf958023239c6A063 | | PoS\-BUSD | PoS | 0x4fabb145d64652a948d72533023f6e7a623c7c53 | 0xdAb529f40E671A1D4bF91361c21bf9f0C9712ab7 | | PoS\-MANA | PoS | 0x0f5d2fb29fb7d3cfee444a200298f468908cc942 | 0xA1c57f48F0Deb89f569dFbE6E2B7f46D33606fD4 | | PoS\-WBTC | PoS | 0x2260fac5e5542a773aa44fbcfedf7c193bc2c599 | 0x1BFD67037B42Cf73acF2047067bd4F2C47D9BfD6 | | PoS\-oxBTC | PoS | 0xb6ed7644c69416d67b522e20bc294a9a9b405b31 | 0x71B821aa52a49F32EEd535fCA6Eb5aa130085978 | | PoS\-KIWI | PoS | 0x2BF91c18Cd4AE9C2f2858ef9FE518180F7B5096D | 0x578360AdF0BbB2F10ec9cEC7EF89Ef495511ED5f | | PoS\-DUST | PoS | 0xbca3c97837a39099ec3082df97e28ce91be14472 | 0x556f501CF8a43216Df5bc9cC57Eb04D4FFAA9e6D | | PoS\-SPN | PoS | 0x20f7a3ddf244dc9299975b4da1c39f8d5d75f05a | 0xeAb9Cfb094db203e6035c2e7268A86DEbeD5BD14 | | DummyERC20Token | PoS | 0xf2F3bD7Ca5746C5fac518f67D1BE87805a2Be82A | 0xeFfdCB49C2D0EF813764B709Ca3c6fe71f230E3e | | DummyERC721Token | PoS | 0x71B821aa52a49F32EEd535fCA6Eb5aa130085978 | 0x6EBEAC13f6403D19C95b6B75008B12fd21a93Aab | | DummyERC1155Token | PoS | 0x556f501CF8a43216Df5bc9cC57Eb04D4FFAA9e6D | 0xA0c68C638235ee32657e8f720a23ceC1bFc77C77 | | WDEV | PoS | 0x4a5dfd63b0c37b38515e4ee51baf40edd420bf7d5 | 0xa5577d1cec2583058a6bd6d5deac44797c205701 | Mumbai (deprecated) | Token name | Bridge | Parent chain address: Goerli | Child chain: Mumbai | | ---------------- | ------ | -- | -- | | PoS\-WETH | PoS | 0x60D4dB9b534EF9260a88b0BED6c486fe13E604Fc | 0xA6FA4fB5f76172d178d61B04b0ecd319C5d1C0aa | | DummyERC20Token | PoS | 0x655F2166b0709cd575202630952D71E2bB0d61Af | 0xfe4F5145f6e09952a5ba9e956ED0C25e3Fa4c7F1 | | DummyERC721Token | PoS | 0x084297B12F204Adb74c689be08302FA3f12dB8A7 | 0x757b1BD7C12B81b52650463e7753d7f5D0565C0e | | DummyERC1155Token | PoS | 0x2e3Ef7931F2d0e4a7da3dea950FF3F19269d9063 | 0xA07e45A987F19E25176c877d98388878622623FA | # port-management.md: This guide provides a detailed list of default ports used across Bor and Heimdall nodes. Understanding these ports is crucial for network configuration and effective communication between nodes. Bor node | Name | Port | Tags | Description | | --------------------- | ----- | ------------------------- | -- | | Network Listening Port | 30303 | Public | Port used by Bor for peer connections and synchronization. | | RPC Server | 8545 | Can be Public, Internal | RPC port for sending transactions and fetching data. Heimdall uses this port to obtain Bor headers. | | WebSocket Server | 8546 | Can be Public, Internal | WebSocket port for real-time updates. | | GraphQL Server | 8547 | Internal | GraphQL port for querying data. | | Prometheus Server | 9091 | Can be Public, Monitoring | Prometheus APIs for Grafana data source. Can be mapped to ports 80/443 via an Nginx reverse proxy. | | Grafana Server | 3001 | Can be Public, Monitoring | Grafana web server. Can be mapped to ports 80/443 via an Nginx reverse proxy. | | Pprof Server | 7071 | Internal, Monitoring | Pprof server for collecting Bor metrics. | | UDP Discovery | 30301 | Can be Public, Internal | Default port for Bootnode peer discovery. | Heimdall node | Name | Port | Tags | Description | | --------------------- | ----- | ------------------------- | -- | | Network Listening Port | 26656 | Public | Port used by Heimdall for peer connections and synchronization. | | RPC Server | 1317 | Can be Public, Internal | RPC port for sending transactions and fetching data. Bor uses this port to interact with Heimdall. | # rewards.md: On the Polygon PoS network, validators stake their MATIC tokens as collateral to help secure the network. In return for their service, they earn rewards. To leverage the PoS network's tokenomics, you'll need to participate in the network either as a validator, or a delegator. To be a validator, you need to run a full validator node and stake MATIC. Also, check the validator responsibilities page. To be a delegator, you need to delegate MATIC to a validator. How are validators incentivized? Polygon allocates 12% of its total supply of 10 billion tokens to fund the staking rewards. This is to ensure that the network is seeded well enough until transaction fees gain traction. These rewards are primarily meant to jump-start the network, while the protocol in the long run is intended to sustain itself on the basis of transaction fees. $Validator Rewards = Staking Rewards + Transaction Fees$ This is allocated in a way to ensure gradual decoupling of staking rewards from being the dominant component of the validator rewards. | Year | Target Stake (30% of circulating supply) | Reward Rate for 30% Bonding | Reward Pool | | ------ | -- | --------------------------- | ------------------------- | | First | 1,977,909,431 | 20% | 312,917,369 | | Second | 2,556,580,023 | 12% | 275,625,675 | | Third | 2,890,642,855 | 9% | 246,933,140 | | Fourth | 2,951,934,048 | 7% | 204,303,976 | | Fifth | 2,996,518,749 | 5% | 148,615,670 + 11,604,170 | Below is a sample snapshot of the expected annual rewards for the first 5 years considering staked supply ranging from 5% to 40% at 5% interval | % of circulating supply staked | 5% | 10% | 15% | 20% | 25% | 30% | 35% | 40% | | ------------------------------ | ---- | --- | --- | ----- | ----- | --- | ------ | ----- | | Annual reward for year | | First | 120% | 60% | 40% | 30% | 24% | 20% | 17.14% | 15% | | Second | 72% | 36% | 24% | 18% | 14.4% | 12% | 10.29% | 9% | | Third | 54% | 27% | 18% | 13.5% | 10.8% | 9% | 7.71% | 6.75% | | Fourth | 42% | 21% | 14% | 10.5% | 8.4% | 7% | 6% | 5.25% | | Fifth | 30% | 15% | 10% | 7.5% | 6% | 5% | 4.29% | 3.75% | Who gets the incentives? Stakers running validator nodes and stakers delegating their tokens toward a validator that they prefer. Validators have the option to charge a commission on the reward earned by delegators. The funds belonging to all stakers are locked in a contract deployed on the Ethereum mainnet. No validator holds custody over delegator tokens. Staking rewards The yearly incentive is fixed â€” irrespective of the overall stake or the target bonding rate in the network, the incentive amount is given out as a reward to all signers periodically. In Polygon, there is an additional element of committing periodic checkpoints to the Ethereum mainnet. This is a major part of the validator responsibilities and they are incentivized to perform this activity. This constitutes a cost to the validator which is unique to a Layer 2 solution such as Polygon. We strive to accommodate this cost in the validator staking reward payout mechanism as a bonus to be paid to the proposer, who is responsible for committing the checkpoint. Rewards minus the bonus is to be shared among all stakers, proposer and signers, proportionally. Encouraging the proposer to include all signatures To avail the bonus completely, the proposer must include all signatures in the checkpoint. Because the protocol desires $2/3+1$ weight of the total stake, the checkpoint is accepted even with 80% votes. However, in this case, the proposer gets only 80% of the calculated bonus. Transaction fees Each block producer at Bor is given a certain percentage of the transaction fees collected in each block. The selection of producers for any given span is also dependent on the validatorâ€™s ratio in the overall stake. The remaining transaction fees flow through the same funnel as the rewards which get shared among all validators working at the Heimdall layer. # rpc-endpoints.md: This guide provides an index of network details for the Polygon Amoy testnet and Polygon PoS mainnet, including their associated RPC and node endpoints. Network details Amoy The Amoy testnet serves as a replica of the Polygon mainnet and is primarily used for testing. Obtain testnet tokens from the faucet. Note that these test tokens hold no real-world value. | Properties | Network details | | --------------- | --- | | Network name | Amoy | | Parent chain | Sepolia | | Chain ID | 80002 | | Gas token | POL | | Gas station | AMOY gas station | | RPC endpoint | https://rpc-amoy.polygon.technology/ | | Node endpoint | wss://rpc-amoy.polygon.technology/ | | Heimdall API | https://heimdall-api-amoy.polygon.technology | | Block Explorer | https://amoy.polygonscan.com/ | !!! info "Additional information" For more details, refer to this JSON data. Mainnet The native token for the Polygon PoS mainnet is MATIC, which is used for transaction fees. | Properties | Network details | | --------------- | --- | | Network name | Polygon | | Parent chain | Ethereum | | Chain ID | 137 | | Gas token | POL | | Gas station | PolygonScan Gas Tracker | | RPC endpoint | https://polygon-rpc.com/ | | Node endpoint | wss://rpc-mainnet.matic.network | | Heimdall API | https://heimdall-api.polygon.technology | | Block explorer | https://polygonscan.com/ | !!! info "Additional information" For more details, refer to the JSON data. RPC API methods Developers can interact with on-chain data and execute various types of transactions using network endpoints. These APIs adhere to the JSON-RPC standard, a stateless, lightweight remote procedure call (RPC) protocol. !!! info "Getting started with RPC calls" For a comprehensive list of API documentation, visit Polygon JSON-RPC calls. To explore API requests without any setup, fix failing requests, or discover new methods on the Polygon network, try the Composer App. Infrastructure providers Public RPCs may have rate limits or traffic restrictions. For dedicated free RPC URLs, consider the following providers: - Alchemy - Allnodes - All That Node - Amazon Managed Blockchain - Ankr - Blast (Bware Labs) - BlockPI - Chainnodes - Chainstack - DataHub (Figment) - Dwellir - GetBlock - Infura - Moralis - NodeReal - OnFinality - QuickNode - SettleMint - Tatum - WatchData - NOWNodes - Kriptonio - Chain49 - Chainbase - Stackup - 1RPC - 4EVERLAND - SubQuery - Validation Cloud - dRPC For a complete list of public endpoints, visit Alchemy's Chain Connect and Chainlist. # seed-and-bootnodes.md: This document provides a comprehensive list of seeds and bootnodes for both the Mainnet and Amoy Testnet, including Bor and Heimdall. !!! info You don't need to configure seeds separately for the Amoy Testnet, as they are already included in the genesis file. However, if you're having trouble connecting to peers, feel free to use them. PoS mainnet !!! info "Selecting a URL" Triple-click a URL to select it, and then copy it using the appropriate keyboard shortcut. Bor bash

enode://b8f1cc9c5d4403703fbf377116469667d2b1823c0daf16b7250aa576bacf399e42c3930ccfcb02c5df6879565a2b8931335565f0e8d3f8e72385ecf4a4bf160a@3.36.224.80:30303
enode://8729e0c825f3d9cad382555f3e46dcff21af323e89025a0e6312df541f4a9e73abfa562d64906f5e59c51fe6f0501b3e61b07979606c56329c020ed739910759@54.194.245.5:30303 Heimdall bash
1500161dd491b67fb1ac81868952be49e2509c9f@52.78.36.216:26656 dd4a3f1750af5765266231b9d8ac764599921736@3.36.224.80:26656 8ea4f592ad6cc38d7532aff418d1fb97052463af@34.240.245.39:26656
e772e1fb8c3492a9570a377a5eafdb1dc53cd778@54.194.245.5:26656 6726b826df45ac8e9afb4bdb2469c7771bd797f1@52.209.21.164:26656 Amoy testnet Bor bash
enode://383ec39eb7f7e23538ea846f502602632110a6bcfc7521bfc2b8833f5a190779507d006b28650d83674b75d188cb36bcb3c3e168a0f2b3d98f9a651cc6603146@52.214.229.208:30303
enode://bce861be777e91b0a5a49d58a51e14f32f201b4c6c2d1fbea6c7a1f14756cbb3f931f3188d6b65de8b07b53ff28d03b6e366d09e56360d2124a9fc5a15a0913d@54.217.171.196:30303
enode://a4a387ad423a2fd0d652808b270082250d3c616b7e8537209584ebad4806dd50ef8dc66a371c85c7f55e6c1f53747edbb11055c8073cfacf312047eaeb328f58@54.171.220.164:30303
enode://4a3dc0081a346d26a73d79dd88216a9402d2292318e2db9947dbc97ea9c4afb2498dc519c0af04420dc13a238c279062da0320181e7c1461216ce4513bfd40bf@13.251.184.185:30303
enode://e8fe33b52f90d4bc7a4e75800945df449d1a091bd347c9f11cd1dbcd98ea28cb4c231cb3b1c6feacdabca2aa91f1a6744724b44edc9382c107968792abdef261@52.74.18.182:30303
enode://b240f1f18e8f3cc61df96a164ba215ea6fc3f00717e4300da6283362a0438bda53f81ecc24c575ff130066d42096319fa027c952681bbb4f003e0bdd5d5b4e61@52.76.37.145:30303
enode://de55d16b6e1fca28cdd3d11eb0dd89e3b77b96d4722172bd5e04ac255922324076a87748e97bc021af2307dccbb5ef8062389cfcba1845f77219eee7935dea9f@52.74.125.36:30303
enode://7f2272685fc3e31c8e43c7687dda43ea3192fd310ba01efcb7811d5dc7ad5a64402ea8cd827650e573a174cf29bb69331dffcca6f0b9894ef17eeafabd97a41d@47.128.184.10:30303
enode://c66e12243b425b63528dd8b1ce87f2f7fbc85f35485e2d8bf6bbf0ec0dcd05b3a582ef62daadbde061b58058735788335d09ed972a451242b9943b85d323c239@63.32.214.97:30303
enode://bd56c0f00dd37e14ae2b84f5eb50e357d3a2d326bdbb0cbb987411268b3f132288f6c86157fc132c6902d18b9be0de8bbdcd12d926e16232ebadd8e274aae780@52.208.81.179:30303
enode://2f015d5b1571165975382281a2117a9b514e1b38e87a8116596fc9b3b121a93cfb238eb6f7b3ae30cf9c0154384372745ce9edc09cbc30526ab7e2059f57ddee@54.74.160.230:30303 Heimdall bash
9b62e5df9711ff0124774295c3a0159a829a9ea8@52.214.229.208:26656 eb57fffe96d74312963ced94a94cbaf8e0d8ec2e@54.217.171.196:26656 b217aaadef7a5f409722e03c5c3b463b48fbb1f3@54.171.220.164:26656
080dcdffcc453367684b61d8f3ce032f357b0f73@13.251.184.185:26656 3fde1fba0bba390fa79bef2393104758b302bf5a@52.74.18.182:26656 7cd4b5fd12c1c7debb7dc9938013b4e9470a5add@52.76.37.145:26656
a4601f540c901112e3c718dd834de5537d8343e8@52.74.125.36:26656 bda8943b47d96ccf8f36dc5d306450926906571b@47.128.184.10:26656 21029acc7239d622019a4213ba9c58aa862b762d@63.32.214.97:26656
46ee7b8d33ade8aa41fd1b96f26a1f275721ee49@52.208.81.179:26656 Alternatively, you can find more seeds here. # delegation.md: Polygon supports delegation via validator shares. By using this design, it is easier to distribute rewards and slash with scale (thousands of delegators) on Ethereum contracts without much computation. Delegators delegate by purchasing shares of a finite pool from validators. Each validator will have their own validator share token. Let's call these fungible tokens VOL for a validator A. VOL refers to validator-specific minted validator share ERC20 tokens. As soon as a user delegates to a validator A, they will be issued VOL based on an exchange rate of POL/VOL pair. As users accrue value the exchange rate indicates that they can now withdraw more POL for each VOL and when users get slashed, users withdraw less POL for their VOL. !!! info POL is a staking token. A delegator needs to have POL tokens to participate in the delegation. Initially, a delegator D buys tokens from validator A specific pool at 1 POL per 1 VOL. When a validator gets rewarded with more POL tokens, new tokens are added to the pool. Let's say with the current pool of 100 POL, 10 POL rewards are added to the pool. But since the total supply of VOL tokens didn't change due to rewards, the exchange rate becomes 1 POL per 0.9 VOL. Now, delegator D gets more POL for the same shares. Technical specification solidity uint256 public validatorId; // Delegation contract for validator uint256 public validatorRewards; // accumulated rewards for validator uint256 public commissionRate; // validator's cut % uint256 public validatorDelegatorRatio = 10; // to be implemented/used uint256 public totalStake; uint256 public rewards; // rewards for pool of delegation stake uint256 public activeAmount; // of tokens delegated which are part of active stake Exchange rate is calculated as below: js ExchangeRate = (totalDelegatedPower + delegatorRewardPool) / totalDelegatorShares Methods and variables buyVoucher js function buyVoucher(uint256 amount) public; - Transfer the amount to stakeManager and update the timeline data structure for active stake. - updateValidatorState is used to update timeline DS. - Mint delegation shares using current exchangeRate for amount. - amountStaked is used to keep track of active stake of each delegator in order to calculate liquid rewards. sellVoucher js function sellVoucher() public; - Using current exchangeRate and number of shares to calculate total amount (active stake + rewards). - unBond active stake from validator and transfer rewards to delegator, if any. - Must remove active stake from timeline using updateValidatorState in stakeManger. - delegators mapping is used to keep track of stake in withdrawal period. withdrawRewards js function withdrawRewards() public; - For a delegator, calculate the rewards and transfer, and depending upon exchangeRate burn count of shares. - Example: if a delegator owns 100 shares and exchange rate is 200 so rewards are 100 tokens, transfer 100 tokens to delegator. Remaining stake is 100 so using exchange rate 200, now it is worth 50 shares. So burn 50 shares. Delegator now has 50 shares worth 100 tokens (which he initially staked / delegated). reStake Restake can work in two ways: delegator can buy more shares using buyVoucher or reStake rewards. js function reStake() public; Above function is used to reStake rewards. The number of shares arenâ€™t affected because exchangeRate is the same; so just the rewards are moved into active stake for both validator share contract and stakeManager timeline. getLiquidRewards is used for calculating accumulated rewards i.e., delegator owns 100 share and exchange rate is 200, so rewards are 100 tokens. Move 100 tokens into active stake, since exchange rate is still same number of share will also remain same. Only difference is that now 200 tokens are considered into active stake and can't be withdrawn immediately (not a part of liquid rewards). Purpose of reStaking is that since delegator's validator has now more active stake and they will earn more rewards for that so will the delegator. unStakeClaimTokens js function unStakeClaimTokens() Once withdrawal period is over, delegators who've sold their shares can claim their POL tokens. Must transfer tokens to user. updateCommissionRate js function updateCommissionRate(uint256 newCommissionRate) external onlyValidator - Updates commission % for the validator. updateRewards js function updateRewards(uint256 reward, uint256 checkpointStakePower, uint256 validatorStake) external onlyOwner returns (uint256) When a validator gets rewards for submitting checkpoint, this function is called for disbursements of rewards between validator and delegators. # genesis-contracts.md: Here you will find a list of contracts deployed on Polygon together with their initial address, that is, their location on the blockchain. Mainnet Parent chain: Ethereum mainnet | Contracts | Address | | --------------------------- | -- | | Governance | 0x98165b71cdDea047C0A49413350C40571195fd07 | | GovernanceI | 0x6e7a5820baD6cebA8Ef5ea69c0C92EbbDAc9CE48 | | Timelock | 0xCaf0aa768A3AE1297DF20072419Db8Bb8b5C8cEf | | Registry | 0x33a02E6cC863D393d6Bf231B697b82F6e499cA71 | | RootChain | 0x536c55cFe4892E581806e10b38dFE8083551bd03 | | RootChainProxy | 0x86E4Dc95c7FBdBf52e33D563BbDB00823894C287 | | ValidatorShareFactory | 0xc4FA447A0e77Eff9717b09C057B40570813bb642 | | StakingInfo | 0xa59C847Bd5aC0172Ff4FE912C5d29E5A71A7512B | | StakingNFT | 0x47Cbe25BbDB40a774cC37E1dA92d10C2C7Ec897F | | StakeManager | 0xbA9Ac3C9983a3e967f0f387c75cCbD38Ad484963 | | StakeManagerProxy | 0x5e3Ef299fDDf15eAa0432E6e66473ace8c13D908 | | SlashingManager | 0x01F645DcD6C796F6BC6C982159B32fAaaebdC96A | | ValidatorShare | 0x01d5dc56ad4206bb0c132d834644d57f51fed5ec | | StateSender | 0x28e4F3a7f651294B9564800b2D01f35189A5bFbE | | DepositManager | 0xDdaC6D3A2a787b1F4bf26AB6FAF519ae3F1a94cf | | DepositManagerProxy | 0x401F6c983eA34274ec46f84D70b31C151321188b | | EventsHubProxy | 0x6dF5CB08d3f0193C768C8A01f42ac4424DC5086b | | WithdrawManager | 0x4ef5123a30e4CFeC02B3E2F5Ce97F1328B29f7de | | ExitNFT | 0xDF74156420Bd57ab387B195ed81EcA36F9fABAca | | WithdrawManagerProxy | 0x2A88696e0fFA76bAA1338F2C74497cC013495922 | | ERC20Predicate | 0x158d5fa3ef8e4dda8a5367decf76b94e7effce95 | | ERC721Predicate | 0x54150f44c785d412ec262fe895cc3b689c72f49b | | EIP1559Burn | 0x70bca57f4579f58670ab2d18ef16e02c17553c38 | | PolygonMigrationProxy | 0x29e7DF7b6A1B2b07b731457f499E1696c60E2C4e | | PolygonMigration | 0x550B7CDaC6F5a0d9e840505c3Df74aC045530446 | | DefaultEmissionManagerProxy | 0xbC9f74b3b14f460a6c47dCdDFd17411cBc7b6c53 | | DefaultEmissionManager | 0x5e875267f65537768435C3C6C81cd313a570B422 | | Tokens | | | MaticToken | 0x7D1AfA7B718fb893dB30A3aBc0Cfc608AaCfeBB0 | | PolygonEcosystemToken | 0x455e53CBB86018Ac2B8092FdCd39d8444aFFC3F6 |

| RootERC721 | 0x96CDDF45C0Cd9a59876A2a29029d7c54f6e54AD3 | | MaticWeth | 0xa45b966996374E9e65ab991C6FE4Bfce3a56DDe8 | Child chain: PoS mainnet | Contracts | Address | | ----------- | -- | | ChildChain | 0xD9c7C4ED4B66858301D0cb28Cc88bf655Fe34861 | | EIP1559Burn | 0x7A8ed27F4C30512326878652d20fC85727401854 | | Tokens | MaticToken | 0x0000000000000000000000000000000000001010 | | MaticWeth | 0x8cc8538d60901d19692F5ba22684732Bc28F54A3 | Helper functions | Function | Address | | ------------------ | -- | | BytesLib | 0x1d21fACFC8CaD068eF0cbc87FdaCdFb20D7e2417 | | Common | 0x31851aAf1FA4cC6632f45570c2086aDcF8B7BD75 | | ECVerify | 0x71d91a8988D81617be53427126ee62471321b7DF | | Merkle | 0x8b90C7633F1f751E19E76433990B1663c625B258 | | MerklePatriciaProof | 0x8E51a119E892D3fb324C0410F11f39F61dec9DC8 | | PriorityQueue | 0x61AdDcD534Bdc1721c91740Cf711dBEcE936053e | | RLPEncode | 0x021c2Bf4d2941cE3D593e07317EC355937bae495 | | RLPReader | 0xD75f1d6A8A7Dc558A65c2f30eBF876DdbeE035a2 | | SafeMath | 0x96D358795782a73d90F2ed2d505aB235D197ca05 | Amoy Parent chain: Sepolia | Contracts | Address | | -------------------------- | -- | | Governance | 0x7ebDeC03873994A02acA5dbfac665e5e39287D77 | | GovernanceProxy | 0xB7086eda3180c728C1536B35c4d54F6A2B33D6aC | | Registry | 0xfE92F7c3a701e43d8479738c8844bCc555b9e5CD | | RootChain | 0xD23E7ebB837de8625F16F575077D176b9d6B9b39 | | RootChainProxy | 0xbd07D7E1E93c8d4b2a261327F3C28a8EA7167209 | | ValidatorShareFactory | 0xF5f09458Ecc701Dd93fa4C88b67d550289Ae6805 | | StakingInfo | 0x5E3111a5d928D24718c1A7897261D0B9087002ed | | StakingNFT | 0x72CF5618142Eb369E75ec6529A907e9A6Fe99bB7 | | StakeManager | 0xE3104cC25C94b21a162d316064fe50fDDA0635aC | | StakeManagerProxy | 0x4AE8f648B1Ec892B6cc68C89cc088583964d08bE | | SlashingManager | 0x9e699267858ce513eACF3b66420334785f9c8E4c | | ValidatorShare | 0xa37F2A3dF304a7C6AbA3492146Bc8B89170bEAb1 | | StateSender | 0x49E307Fa5a58ff1834E0F8a60eB2a9609E6A5F50 | | DepositManager | 0x7270E1fa2f0569d50dCd1D038D940135bDE6167F | | DepositManagerProxy | 0x44Ad17990F9128C6d823Ee10dB7F0A5d40a731A4 | | WithdrawManager | 0xE1D0DdD817CA39301c9E4F5A7218BbDAf59D248D | | ExitNFT | 0x68EB9202b48D3980832aDE89d580bA883b313B40 | | WithdrawManagerProxy | 0x822db7e79096E7247d9273E5782ecAec464Eb96C | | ERC20Predicate | 0x15EA6c538cF4b4A4f51999F433557285D5639820 | | ERC721Predicate | 0x0059bBF8E5b9b071acc7682B6fe198c32AAA2A97 | | EventsHubProxy | 0x700e0f2AfBd92e2b3fF91CAD8C62A564690ddf39 | | EIP1559Burn | 0xeCDD77cE6f146cCf5dab707941d318Bd50eeD2C9 | | PolygonMigrationProxy | 0x3A3B750E7d4d389Bc1d0be20E5D09530F82B9911 | | PolygonMigration | 0xC70198ad91082c4d6eEb70d991cc4B2b61Cb3d1E | | DefaultEmissionManagerProxy | 0x20393fF3B3C38b72a16eB7d7A474cd38ABD8Ff27 | | DefaultEmissionManager | 0x9B4D4Fc98C2Aa924a65AB827494159a76Fce6ceD | | Tokens | | | MaticToken | 0x3fd0A53F4Bf853985a95F4Eb3F9C9FDE1F8e2b53 | | PolygonEcosystemToken | 0x44499312f493F62f2DFd3C6435Ca3603EbFCeeBa | | MaticWeth | 0x700dDE29De87ed2c01c27C896dc8Badb4f671302 | | RootERC721 | 0x13B0Edd9312886Ac0C73116e767208bEd1199679 | Child chain: Amoy | Contracts | Address | | ----------- | -- | | ChildChain | 0x4f9cd8a945EE035523979D7A120a23999D17D8C0 | | EIP1559Burn | 0xeCDD77cE6f146cCf5dab707941d318Bd50eeD2C9 | | Tokens | | | MaticToken | 0x0000000000000000000000000000000000001010 | | MaticWeth | 0x41Dc3C8eB8368bd9139Cec50434a0C294c8c1102 | | RootERC721 | 0x3ADBC484Ff0cFEb657e1A9AF8F3CB16DC0B53e7e | # stakingmanager.md: For the Polygon PoS Proof of Security based consensus, all the $2/3+1$ proof verification and handling of staking, rewards are executed on the Ethereum smart contract. The whole design follows this philosophy of doing less on the Mainnet contract. It does information verification and pushes all the computation-heavy operations to L2 (read about Heimdall). Stakers are divided into validators, delegators, and watchers (for fraud reporting). StakeManager is the main contract for handling validator related activities like checkPoint signature verification, reward distribution, and stake management. Since the contract is using NFT ID as a source of ownership, change of ownership and signer won't affect anything in the system. !!! info From a single Ethereum address, a staker can only function as either a validator or a delegator (this is a design choice without any underlying technical constraints). Validator admissions/replacement Admissions Currently, the system allows for a maximum of 105 active validators at any given time on Polygon PoS. There is also a waitlist to become a validator. New validators can join the active set only when a currently active validator unbonds or is removed due to low performance. You can apply to become a validator via the Polygon validators hub. Please note that submitting an application does not guarantee a validator slot. More relevant information is available on the becoming a validator page. Replacement PIP4 introduced the concept of showcasing validator performance for community visibility. If a validator is in an unhealthy state for an extended period of time as outlined in PIP4, they are off-boarded from the network. The validator slot is then made available to those coming off of the waitlist. Learn more about these requirements on the validator performance requirements page. Methods and variables validatorThreshold It stores the maximum number of validators accepted by the system, also called slots. AccountStateRoot - For various accounting done on Heimdall for validators and delegator, account root is submitted while submitting the checkpoint. - accRoot is used while claimRewards and unStakeClaim. stake / stakeFor solidity title="StakeManager.sol" function stake(uint256 amount, uint256 heimdallFee, bool acceptDelegation, bytes calldata signerPubkey) public; function stakeFor(address user, uint256 amount, uint256 heimdallFee, bool acceptDelegation, bytes memory signerPubkey) public; - Allows anyone with amount (in MATIC tokens) greater than minDeposit, if currentValidatorSetSize is less then validatorThreshold. - Must transfer amount+heimdallFee. - updateTimeLine updates special timeline data structure, which keeps track of active validators and active stake for given epoch / checkpoint count. - One unique NFT is minted on each new stake or stakeFor call, which can be transferred to anyone but can be owned 1:1 Ethereum address. - acceptDelegation set to true if validators want to accept delegation, ValidatorShare contract is deployed for the validator. unstake - Remove validator from validator set in next epoch (only valid for current checkpoint once called unstake) - Remove validator's stake from timeline data structure, update count for validator's exit epoch. - If validator had delegation on, collect all rewards and lock delegation contract for new delegations. unstakeClaim solidity function unstakeClaim(uint256 validatorId) public; Once WITHDRAWALDELAY period is served, validators can call this function and do settlement with stakeManager (get rewards if any, get staked tokens back, burn NFT, etc). restake solidity function restake(uint256 validatorId, uint256 amount, bool stakeRewards) public; - Allows validators to increase their stake by putting new amount or rewards or both. - Must update timeline (amount) for active stake. withdrawRewards solidity function withdrawRewards(uint256 validatorId) public; This method allows validators to withdraw accumulated rewards, must consider getting rewards from delegation contract if validator accepts delegation. updateSigner solidity function updateSigner(uint256 validatorId, bytes memory signerPubkey) public This method allows validators to update signer address (which is used to validate blocks on Polygon blockchain and checkpoint signatures on stakeManager). topUpForFee solidity function topUpForFee(uint256 validatorId, uint256 heimdallFee) public; Validators can top-up their balance for Heimdall fee by invoking this method. claimFee solidity function claimFee(uint256 validatorId, uint256 accumSlashedAmount, uint256 accumFeeAmount, uint256 index, bytes memory proof) public; This method is used to withdraw fees from Heimdall. accountStateRoot is updated on each checkpoint, so that validators can provide proof of inclusion in this root for account on Heimdall and withdraw fee. Note that accountStateRoot is re-written to prevent exits on multiple checkpoints (for old root and save accounting on stakeManager). Staking NFT Standard ERC721 contract with few restrictions like one token per user and minted in sequential manner. checkSignatures solidity function checkSignatures(uint256 blockInterval, bytes32 voteHash, bytes32 stateRoot, bytes memory sigs) public; - Writes are meant only for RootChain contract when submitting checkpoints - voteHash on which all validators sign (BFT $2/3+1$ agreement) - This function validates only unique sigs and checks for $2/3+1$ power has signed on checkpoint root (inclusion in voteHash verification in RootChain contract for all data) currentValidatorSetTotalStake provides current active stake. - Rewards are distributed proportionally to validator's stake. More on rewards on the rewards distribution page. isValidator Checks if a given validator is active validator for the current epoch. Timeline data structure solidity struct State { int256 amount; int256 stakerCount; } mapping(uint256 => State) public validatorState; !Figure: Knowledge base - node setup 1 StakingInfo Centralized logging contract for both validator and delegation events, includes few read only functions. You can check out the source code of the StakingInfo.sol contract on GitHub. ValidatorShareFactory A factory contract to deploy ValidatorShare contract for each validator who opt-in for delegation. You can check out the source code of the ValidatorShareFactory.sol contract on GitHub. # index.md: --- hide: - toc ---

Developer tools

This section describes some of the in-house and third-party tools that are used by developers to work with Polygon products and services.

Find out how to access data, code against blockchain networks, use oracles, and much more.

All third-party content in this section is covered by our content disclaimer.

dApp Launchpad

Automated CLI tool for initializing, creating, and deploying fully-integrated, web3 dApp projects.

Chain indexer framework

Web3 data indexing framework for developing flexible, event-driven data pipelines on EVM blockchains.

Token faucets

Different ways to fetch test POL and ETH on Polygon PoS and zkEVM network.

MaticJS

The matic.js library used to interact with Polygon networks and services.

Polygon Portal

Polygon's asset management dashboard used for bridging and claiming.

Wallets

Using Polygon-compatible external wallets, such as MetaMask, with Polygon networks.

Block explorers

Links to useful explorers like Polygonscan and OKX explorer.

Storage

Interact with blockchain storage services such as IPFS.

Oracles

Oracle services used for accessing accurate offline data.

overview.md: !!! info "Git repo and contact details" - For details about the technical architecture and if you want to contribute, head over to the repo https://github.com/0xPolygon/chain-indexer-framework - For any questions, reach out to us on the Polygon R&D Discord. Overview & problem statement The Chain Indexer Framework is a powerful blockchain data indexer framework designed for the development of flexible event-driven data pipelines on EVM blockchains. Built on the reliable foundation of Kafka,Â the Chain Indexer Framework empowers developers to build robust and scalable applications that seamlessly process blockchain events and enable real-time data integration. The Chain Indexer Framework revolutionizes the way developers interact with blockchain data, offering a fast, secure, and efficient method for data retrieval. By choosing the Chain Indexer Framework you are not just selecting a tool, but opting for a more streamlined and efficient development process for your dApps. Before we get into the details, let us understand why we need it and the problem it solves. 1. What is an EVM blockchain data indexer? An EVM Ethereum Virtual Machine blockchain data indexer is a specialised search engine for EVM-based blockchains like Ethereum and Polygon protocols. It sorts and organises blockchain data, making it faster and easier for developers to query and retrieve specific information for their decentralised applications (dApps). 2. Why do dApps need a blockchain data indexer? Without a data indexer like Chain Indexer Framework, querying blockchain data can be cumbersome and slow. Traditional methods require developers to sift through enormous volumes of transaction records, smart contracts, and other types of data. This is like trying to find a single paragraph in an entire library of uncategorized books. Chain Indexer Framework makes this process fast and efficient, acting as a digital librarian that quickly retrieves the specific data the user is looking for. !!! info "An helpful analogy" Imagine that a blockchain is like a massive public library, and each book in the library represents a block of transactions. Now, think of an EVM-based dApp as a researcher who needs specific pieces of information from multiple books in this huge library for their project. Without a catalog or librarian, this researcher would spend an enormous amount of time locating what they need. A blockchain data indexer like Chain Indexer Framework acts as both the catalog system and the super-efficient librarian for this library. It knows exactly where each "book" is located and can instantly pinpoint the specific "page" you need. The indexer saves the researchers (dApps) valuable time and resources, allowing them to focus on their main project instead of getting bogged down with the cumbersome task of data retrieval. 3. Why is it hard to build a blockchain data indexer? The development complexity in building a blockchain data indexer arises from the need to integrate a diverse software stack that often includes databases, queueing systems, and caching layers. Ensuring real-time performance, handling data consistency during blockchain forks, and adapting to protocol changes add to the complexity. Rigorous testing is essential to validate the indexer under various scenarios, making it a challenging endeavour that requires expertise in multiple domains. Things get more complicated in the multi-chain world. The solution: Chain Indexer Framework The Chain Indexer Framework is a powerful developer framework designed to index raw blockchain data, which can later be utilized by developers to build the backend layers of their decentralized applications (dApps) based on their specific dApp logic. Think of the Chain Indexer Framework as a tool that helps you construct the "Search Engine" for blockchain data; it indexes, categorizes, and makes data to be quickly and cost-effectively accessible to developers. Chainflow brings the power of advanced data indexing to dApps, simplifying and accelerating the process of fetching the data developers need. !!! note The Chain Indexer Framework is a tool that gathers information from a blockchain and channels it into a data stream known as Kafka. However, such data is raw, and not yet ready for use. It must therefore be transformed and stored in a database like Postgres or MongoDB. Only then can a decentralized application (dApp) utilize it. Chainflow provides the framework/tools that enable developers to transform the raw data based on their individual dApp requirements. In short, the Chain Indexer Framework handles the tedious task of collecting and preparing blockchain data for developers. Afterward, developers will need to build specific features for their app, such as how to use the data. Chain Indexer Framework offers foundational logic and helper functions that make the task much easier. How does Chain Indexer Framework work? Chain Indexer Framework employs a combination of caching, distributed architecture, and advanced algorithms to quickly retrieve the data a dApp might require. It takes raw blockchain data, indexes it, and helps to convert it into easily accessible formats that developers can query using simple APIs. Initially, Chain Indexer

Framework obtains the raw blockchain data via blockchain RPC and stores it in a Kafka stream. It provides the core logic and helper functions needed to transform this raw data. Additionally, it allows you to consume the data from the Kafka stream and stores it in a separate database. Developers can then host the APIs for their dApp on top of this database. Features & benefits Why choose Chain Indexer Framework? 1. Open source: Anyone can fork, modify, and host the software on their own infrastructure. Unlike common third-party indexers, Chain Indexer Framework offers complete control to developers, enabling them to monitor the service closely in order to identify and resolve issues. 2. Cost savings: developers can save money otherwise spent on third-party data indexers. There will be no restrictions on usage or API rate limits, as developers will host the service themselves. 3. Built using TypeScript: This developer-friendly programming language makes it easy to understand the core logic and implement changes. Chain Indexer Framework can be easily installed by downloading the NPM package and integrating it into your project. 4. One-time effort for indexing historical blocks: Once the raw blockchain data is indexed, developers can build an unlimited number of application layers on top of it. There will be no need to make RPC calls to re-index historical blocks, as they will already be available in the Kafka Data Warehouse. 5. Modular architecture: Chain Indexer Framework features a modular architecture, simplifying debugging and bug-fixing processes for developers. 6. Instant query: Imagine a customer walking into your retail store and asking for a specific product. If your inventory is well-organized, you can instantly determine whether the item is in stock and where it is located. Chain Indexer Framework enables similar real-time data queries, making it easier for dApps to access and utilize blockchain data. 7. Event-triggered actions: Consider how online stores send customers notifications when an item is back in stock. Chain Indexer Framework can establish event triggers for specific blockchain activities, allowing dApps to automatically execute actions such as sending notifications or updating the user-interface. 8. Customization: Just as some stores need to track perishable items differently from non-perishable ones, dApps often have unique data requirements. Chain Indexer Frameworkâ€™s flexible architecture allows developers to customize data pipelines to meet their specific needs. 9. Scalability: A small retail store might initially work with manual sorting, but as the business expands, automation becomes essential. Chain Indexer Framework can scale alongside your dApp's growth, handling increased data loads without sacrificing performance. 10. Improved User Experience: Nothing frustrates a customer more than slow or unresponsive service. By providing faster and more reliable access to blockchain data, Chain Indexer Framework helps dApps offer a smoother, more responsive user experience. Popular use cases 1. Wallet services: Blockchain indexers can help wallet providers offer more features like transaction history, balance history, and real-time updates. 2. dApp backend: dApps often require real-time access to contract events, token transactions, and other on-chain activities. A data indexer can speed up this process considerably. 3. Analytics and monitoring: Firms specializing in blockchain analytics use indexers to monitor activities like fraudulent transactions, smart contract interactions, and trends in token transfers. 4. Cross-chain services: For cross-chain swaps or interactions, indexers can offer data that facilitates more seamless integrations. 5. Oracles: Data indexers can support oracles by providing them with a more efficient way to access specific data points on the blockchain. 6. NFT marketplaces: To track ownership changes, price histories, and various attributes of NFTs, data indexers are often used in the backend. # usage.md: !!! info "Git repo and contact details" - For details about the technical architecture and if you want to contribute, head over to the repo https://github.com/0xPolygon/chain-indexer-framework - For any questions, reach out to us on the Polygon R&D Discord. Installation You can install the package using: - NPM - Yarn Using npm bash npm install @maticnetwork/chain-indexer-framework Using Yarn bash yarn add @maticnetwork/chain-indexer-framework Usage jsx // Import the chain-indexer-framework module const chain-indexer-framework = require('@maticnetwork/chain-indexer-framework'); Examples To gain a clearer understanding of the entire process, check out our prebuilt examples. - The first example involves indexing MATIC transfer events from the Ethereum blockchain. - The second example involves indexing NFT transfers and maintaining NFT balances. Both these examples encompass all the layers involved, starting from producers, moving through transformers, and concluding with consumers. # third-party-tutorials.md: !!! warning "Content disclaimer" Please view the third-party content disclaimer here. Explore the categories below to discover third-party resources and tutorials to support your journey in developing, deploying, and managing dApps in the Polygon ecosystem. Node and infra providers - Alchemy - ChainStack - GetBlock PoS - GetBlock zkEVM - QuickNode PoS - QuickNode zkEVM Dev tools and platforms - ChainIDE - EthCreator - Openfort - Thirdweb Chain data and analytics services - OKX explorer - Verify contract with API - Verify contract with external plugins Minting and payment solutions - Venly - NFT API - Zapier integration - Crossmint # hardhat.md: Overview Hardhat is an Ethereum development environment for deploying smart contracts, running tests, and debugging Solidity code locally. In this tutorial, you will learn how to set up Hardhat and use it to build, test, and deploy a simple smart contract. Set up 1. Ensure you have installed the following: - Node.js v10+ LTS and npm. - Git. 2. Create an npm project sh mkdir hardhat-test cd hardhat-test/ npm init 3. Now install Hardhat. sh npm install --save-dev hardhat !!! note The sample project used here comes from the Hardhat Quickstart guide, as well as its instructions. Creating a project To create a sample project, run npx hardhat in your project folder. You should see the following prompt: !img Choose the JavaScript project and go through these steps to compile, test and deploy the sample contract. Checking the contract The contracts folder contains Lock.sol, which is a sample contract which consists of a simple digital lock, where users could only withdraw funds after a given period of time. solidity // SPDX-License-Identifier: UNLICENSED pragma solidity ^0.8.9; // Import this file to use console.log import "hardhat/console.sol"; contract Lock { uint public unlockTime; address payable public owner; event Withdrawal(uint amount, uint when); constructor(uint unlockTime) payable { require(block.timestamp < unlockTime, "Unlock time should be in the future"); unlockTime = unlockTime; owner = payable(msg.sender); } function withdraw() public { // Uncomment this line to print a log in your terminal // console.log("Unlock time is %o and block timestamp is %o", unlockTime, block.timestamp); require(block.timestamp >= unlockTime, "You can't withdraw yet"); require(msg.sender == owner, "You aren't the owner"); emit Withdrawal(address(this).balance, block.timestamp); owner.transfer(address(this).balance); } } Setting up the contract - Go to hardhat.config.js - Update the hardhat-config with matic-network-credentials - Create .env file in the root to store your private key - Add Polygonscan API key to .env file to verify the contract on Polygonscan. You can generate an API key by creating an account js require('dotenv').config(); require("@nomiclabs/hardhat-ethers"); require("@nomiclabs/hardhat-etherscan"); module.exports = { defaultNetwork: "polygonamoy", networks: { hardhat: { }, polygonamoy: { url: "https://rpc-amoy.polygon.technology", accounts: [process.env.PRIVATEKEY] } }, etherscan: { apiKey: process.env.POLYGONSCANAPIKEY }, solidity: { version: "0.8.9", settings: { optimizer: { enabled: true, runs: 200 } } }, } !!! note Note that the file above requires DOTENV, for managing environment variables and also ethers and etherscan. Make sure to install all those packages. Find more instructions on how to use DOTENV on this page. Compiling the contract To compile the contract, you first need to install Hardhat: bash npm install --save-dev @nomicfoundation/hardhat-toolbox Then, simply run to compile: bash npx hardhat compile Testing the contract To run tests with Hardhat, you just need to type the following: bash npx hardhat test And this is an expected output: !img Deploying on Polygon network Run this command in root of the project directory: bash npx hardhat run scripts/deploy.js --network polygonamoy The contract will be deployed on Polygon Amoy testnet, and you can check the deployment status here: Congratulations! You have successfully deployed Greeter Smart Contract. Now you can interact with the Smart Contract. !!! tip "Quickly verify contracts on Polygonscan" Run the following commands to quickly verify your contract on Polygonscan. This makes it easy for anyone to see the source code of your deployed contract. For contracts that have a constructor with a complex argument list, see here. bash npm install --save-dev @nomiclabs/hardhat-etherscan npx hardhat verify --network polygonamoy 0x4b75233D4FacbAa94264930aC26f9983e50C11AF # remix.md: Overview This tutorial guides you to implement a Hello World dApp which echoes a message passed to the contract on to the frontend. You can also change the message using the interactive panel. !!! tip "Recommended" We recommend you to follow this tutorial using the online IDE available atÂ Remix IDE. Remix IDE is an easy-to-use platform that does not require any downloads, creating accounts, or logins. What you will do - Create a file on Remix. - Upload a pre-built smart contract into the IDE. - Compile the smart contract. - Connect the application to the Polygon Amoy testnet via MetaMask. - Deploy the smart contract. - Verify the smart contract. Getting started with Remix IDE Remix is a Ethereum-focused IDE: an online platform to develop and deploy smart contracts. To start building a smart contract, click on New File and name it HelloWorld.sol. `![](../../../img/tools/remix/new-file.png)` Smart contract Copy and paste the smart contract code provided below into the newly created HelloWorld.sol file. js title="HelloWorld.sol" // Specifies that the source code is for a version // of Solidity greater than 0.5.10 pragma solidity ^0.5.10; // A contract is a collection of functions and data (its state) // that resides at a specific address on the Ethereum blockchain. contract HelloWorld { // The keyword "public" makes variables accessible from outside a contract // and creates a function that other contracts or SDKs can call to access the value string public message; // A special function only run during the creation of the contract constructor(string memory initMessage) public { // Takes a string value and stores the value in the memory data storage area, // setting message to that value message = initMessage; } // A publicly accessible function that takes a string as a parameter // and updates message function update(string memory newMessage) public { message = newMessage; } } The first lineÂ pragma solidity ^0.5.10Â specifies that the source code is for a Solidity version greater than 0.5.10.Â PragmasÂ are common instructions for compilers about how to treat the source code (e.g., pragma once). A contract in the sense of Solidity is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain. The lineÂ string public messageÂ declares a public state variable calledÂ messageÂ of typeÂ string. You can think of it as a single slot in a database that you can query and alter by calling functions of the code that manages the database. The keyword public automatically generates a function that allows you to access the current value of the state variable from outside of the contract. Without this keyword, other contracts have no way to access the variable. TheÂ constructorÂ is a special function run during the creation of the contract and cannot be called afterward. In this case, it takes a string valueÂ initMessage, stores the value in theÂ memoryÂ data storage area, and setsÂ messageÂ to that value. TheÂ updateÂ function is another public function that is similar to the constructor, taking a string as a parameter, and updating theÂ messageÂ variable. Compile smart contract - Go to the Solidity Compiler tab (below the search button) - Select compiler version to 0.5.10 - Now, compile HelloWorld.sol - After successful compilation, it will show a green tick mark on the Compiler tab button Deploying to the Amoy testnet Now, we have to deploy our smart contract on Amoy, the Polygon testnet. Not only does it cost money (e.g., gas fees) to deploy a smart contract on Polygon mainnet, but the contract is immutable and can't be changed once deployed. Therefore, it's best to deploy your smart contract to the testnet first. To deploy to the Amoy testnet, we have to connect to the web3 with a service like MetaMask, Brave, Portis, etc. We use MetaMask in this tutorial. Please follow this guide to set up a MetaMask account. - Open MetaMask. Click on the network dropdown menu (set to Ethereum Mainnet by default) and click on the Add Network button. MaticVigil provides a public endpoint and is rate-limited. Therefore, most developers use a free blockchain provider like Alchemy or Quicknode to have a dedicated RPC endpoint with faster throughput AKA computing units per second (CUPs). You need to create a free account to get your private API key from one of these providers to put into the dedicated RPC endpoint URL below. - Network: Polygon Amoy testnet - RPC URL (public endpoint): . - RPC URL (dedicated endpoint): . You need a free API key - Chain ID: 80002 - Currency Symbol: MATIC - Block explorer URL: - Go ahead and click Save - Copy your wallet address from MetaMask by clicking over your account name - Head over to the faucet and request test MATIC to pay for gas on the Polygon network. Select Amoy as the network and MATIC Token as the token in the faucet. You can also use the Alchemy Amoy faucet. - Finally, to deploy to Amoy, refer to the instructions in the Remix deployment section below. Deploying to Polygon mainnet After deploying on Amoy without any errors, you can deploy the smart contract on the Polygon mainnet. Remember this costs real $ in the form of MATIC tokens. 1. Open your MetaMask wallet. 2. Click on the network dropdown menu (set to Ethereum Mainnet by default) and click on the Add Network button. 3. Make sure to have your Alchemy API key to hand. 4. Add the info: - Network Name: Polygon Mainnet - New RPC URL: - Chain ID: 137 - Currency Symbol: MATIC - Block Explorer URL: - Go ahead and click Save - Copy your wallet address from MetaMask by clicking over your account name 5. Make sure your MetaMask wallet has MATIC tokens to pay the deployment's gas fees. Remix deployment In both Testnet and Mainnet, you do the below to deploy your smart contract using Remix. The below step will use the connect MetaMask API keys you set up in the previous steps. - Select Injected Provider MetaMask in the Environment dropdown and your contract !RemixIDEStep1 - Accept the Connect request received in MetaMask. If the popup doesn't open by default, you can also try manually launching the MetaMask extension - Once MetaMask is connected to Remix, the Deploy transaction would generate another MetaMask popup that requires transaction confirmation. Simply confirm the transaction! !RemixIDEStep1

Congratulations!Â You have successfully deployed the HelloWorld smart contract to the Polygon Amoy testnet. You can now start interacting with your smart contract. Check the deployment status at . Verifying your contract Flatten your smart contract The first and foremost step is to flatten the solidity smart contract into a single file. In order to do that, install truffle-flattener or sol-merger. After installation, flatten the contract using below command (we have demonstrated using sol-merger). sh sol-merger \"./contracts/.sol\" ./build Verifying on Polygonscan - Navigate to your contract's Polygonscan page and click on Verify and Publish. !RemixIDEStep1 - Select Solidity (Single File) in compiler type - Select appropriate compiler version - Choose the license type of your contract In the next section, paste your flattened smart contract here. If you had enabled optimization, then adjust the optimization section accordingly. Constructor arguments should have been filled in automatically. If not, they can be retrieved from the trailing bytes of the deployment transaction (example: 000000000000000000000000a6fa4fb5f76172d178d61b04b0ecd319c5d1c0aa). # replit.md: Overview Replit is a coding platform that allows you to write code and host apps. Replit supports Solidity so it provides all of the features and functionality for Web3 developers to create and deploy smart contracts. This article guides you to build and deploy a solidity smart contract on Polygon using Replit IDE and Replit Solidity development template (Solidity starter beta). What you will do - Create a Replit account. - Create a Repl environment. - Deploy a sample project on the Polygon Amoy network. - Verify the contract. - Publish your project to a personal Replit profile. !!! tip For additional examples about Solidity with Replit, you can read the article Get started with Replit. Prerequisites You do not need any local environment setup to deploy your solidity smart contract on Polygon using Replit. You need a browser-based web3 wallet to interact with the Polygon Amoy testnet and deployed contracts. If you are already using MetaMask, we recommend creating a new account for testing with Replit. You can do this from the account menu, which appears when you click on the account avatar in the top right corner of the MetaMask interface. You must set up all of the following prerequisites to be able to deploy your solidity smart contract on Polygon: 1. Create a Replit account. 2. Download MetaMask wallet. 3. Configure Polygon on MetaMask. 4. Get testnet tokens. Working with a Repl Every Repl that you create is a fully functional development and production environment. Follow the steps to create a solidity starter Replit: 1. Log in or create an account. After creating your account, your home screen will include a dashboard where you can view, create projects, and manage your account. !img 2. Once logged in, create a Solidity starter repl, Select + Create Repl from the left panel or + in the top right corner of the screen. !img 3. Select the Solidity starter (beta) template and give your project a title. 4. Click on + Create Repl to create your project. !!! note The Solidity starter repl comes with a browser-friendly interface, built using the Web3 Ethereum JavaScript API, which you can use to deploy and interact with our contracts. We will deploy to Replitâ€™s testnet, a custom version of the Ethereum blockchain managed by Replit and optimised for testing. Deploy on Polygon Make sure that you have followed the list of prerequisites above so that you are ready to deploy and interact with your smart contract. 1. Click on Run (at the top) to install all relevant packages and start up the contract deployment UI. 2. Connect your MetaMask wallet to the web interface and switch to the Amoy testnet. !img 3. Click on Connect wallet, select your account, then choose Connect. !img 4. From the dropdown list, select the contract that you want to deploy. Click on Deploy. 5. You will see a MetaMask popup window asking for your confirmation. Approve the transaction from your wallet to deploy your contract. Verifying and testing your contract When the contract is deployed, navigate to Polygonscan to search for your account, view your deployed contract, and copy your account address. Once your contract has been deployed, it will show up as expandable boxes below the dropdown box. Expand it and take a look at all the different functions available. You can now interact with your contract using the provided user interface or from a sharable URL shown on the interface. Publish to Replitâ€‹ Replit allows you to publish your projects to a personal profile. After publishing, projects will show up on your spotlight page for others to explore, interact with, clone, and collaborate. Follow the below steps to publish your projects to Replit: 1. Select the project title at the top of the screen. 2. Complete your project name and description and click Publish. # commands.md: init Initializes a new project in a new directory which has the same name as the project. If a project name is not mentioned, a random one is chosen. sh dapp-launchpad init [PROJECT-NAME] Options | Option | Description | |:---------------------:|:---------------------------------------| | -t, --template [NAME] | Name of the scaffold template to use; default: "javascript". To get list of available templates, run list scaffold-templates. (default: "javascript") | | -h, --help | display help for command | Help sh dapp-launchpad init -h dev Starts a local dev environment; a local blockchain (Hardhat) and a local front end (Next.js) server. sh dapp-launchpad dev [options] The dev command starts Options | Option | Description | |:---|:---| | -n, --fork-network-name [NAME] | Name of the network to fork; optional. By default, it starts new chain from genesis block. (choices: "ethereum", "goerli", "polygonPos", "polygonAmoy", "polygonZkevm", "polygonZkevmTestnet") | | -b, --fork-block-num [number] | Block number to fork at. By default, it's the latest block. | | -r, --reset-on-change | Resets the entire local blockchain when any code is changed; for forked mode, it resets back to forked block number; NOT DEFAULT. | | --only-smart-contracts | Deploys only smart contracts (having started local test chain) and updates Smart contract configs for frontend; does not start frontend dev environment. | | --only-frontend | Deploys only frontend (having started local server); does not start local blockchain. Smart contracts data is read from pre-existing configs. To generate these manually, use generate smart-contracts-config. | | -e, --enable-explorer | Sets up a chain explorer for the local test blockchain started; NOT DEFAULT; sign up at https://app.tryethernal.com/. | | --ethernal-login-email [EMAIL] | Ethernal login email; needed only if --explorer is enabled. This overrides env variable ETHERNALEMAIL if present. | | --ethernal-login-password [PASSWORD] | Ethernal login password; needed only if --explorer is enabled. This overrides env variable ETHERNALPASSWORD if present. | | --ethernal-workspace [WORKSPACE] | Ethernal workspace name; needed only if --explorer is enabled. This overrides env variable ETHERNALWORKSPACE if present. | | -h, --help | Display help for command | Help sh dapp-launchpad dev -h deploy The deploy command deploys the smart contracts and frontend app to production. sh dapp-launchpad deploy -n CHAINNAME Options | Option | Description | |:-----------------------------|:---| | -n, --network-name | Name of the network to deploy smart contracts to. (choices: "ethereum", "goerli", "polygonPos", "polygonAmoy", "polygonZkevm", "polygonZkevmTestnet") | | --only-smart-contracts | Deploys only smart contracts and updates Smart contracts config for frontend. | | --only-frontend | Deploys only frontend; smart contracts data is read from Smart contracts config which must pre-exist. To generate these manually, use generate smart-contracts-config | | -h, --help | Display help for command | Help sh dapp-launchpad deploy -h list List options. sh dapp-launchpad list scaffold-templates List the available scaffold template languages. sh dapp-launchpad list scaffold-templates generate Generate the specified. sh dapp-launchpad generate smart-contracts-config Generate the smart contract configuration. sh dapp-launchpad generate smart-contracts-config Options | Option | Description | |:-------------------------:|:---------------------------------------| | -e, --environment | Environment where this config would be used (choices: "development", "production", default: "development") | | -n, --network-name | Name of the network to generate config for. (choices: "ethereum", "goerli", "polygonPos", "polygonAmoy", "polygonZkevm", "polygonZkevmTestnet") | | -h, --help | Display help for command | # common-pitfalls.md: MetaMask transactions fail with a nonce error in dev environment Every time the dev environment starts, a new local test chain is started. Metamask maintains a cache of "latest block number" and "account transaction nonce". Since every run of dev creates a new chain, it never matches with this cache. To clear the cache, follow the MetaMask documentation. MetaMask transactions fail with a nonce error when using the reset on change option in dev environment The reset on change option resets the blockchain on every code change. MetaMask maintains a cache of latest block number and account transaction nonce. After resetting the chain, the latest block number and account transaction nonce should go back to the initial state as well, but MetaMask does not update this cache on its own. To clear the cache, follow the MetaMask documentation. Error: could not coalesce error (error={ "code": -32603, "message": "Internal JSON-RPC error." }) Clear your wallet cache and try again. To clear the cache, follow the MetaMask documentation. # contributing.md: Build from source To build from source, clone the dApp Launchpad repository. sh git clone https://github.com/0xPolygon/dapp-launchpad.git cd dapp-launchpad Then build the CLI tool. sh npm run build This generates cli.js inside the bin directory, which can then be installed globally with: sh npm run install-global Now dapp-launchpad is available as a global command. Develop on local To modify this tool, start a local dev environment by running: sh npm run dev This running process watches the source files, bundles up the CLI app on every change, and installs it globally. This means that the app is continuously updating with changes to the code. Report a bugs/request a feature Create an issue below on the repo: https://github.com/0xPolygon/dapp-launchpad/issues Please give as much detail about your issue or request as possible. # frontend.md: Framework The frontend runs on a Next.js server. To get started, modify the component file at ./frontend/src/pages/index. !!! info If you're new to Next.js but know React.js, getting used to Next.js is trivial. To learn more about Next.js, read the Next.js docs. Environment variables Make sure you have followed the steps in the quickstart. !!! note All environment variable names exposed in client requests should be prefixed with NEXTPUBLIC. Connecting wallets To connect a wallet, Web3Modal v3 has been integrated and pre-configured for you. Use the provided useWallet hook to interact with Web3Modal and wallets. This contains utilities to simplify anything you need related to wallets. Sending transactions to smart contracts To send transactions to either a locally deployed smart contract or a smart contract on a prod chain, use the useSmartContract hook. This contains utilities that simplify getting and interacting with a Ethers.js contract instance. When deploying to local or production, this hook automatically uses the correct chain and contracts. Deploying to local test server The dev command automates everything needed for setting up a local Next.js test server. Deploying to Vercel We use Vercel for deployments. Vercel offers free quotas to developers to get started. To deploy, follow the deployment steps. With the deploy command, the frontend deployment is fully automated. No pre-configuration is necessary for running the deploy command. You'll be taken through all relevant steps upon running it. # intro.md: dApp Launchpad is an automated CLI tool for initializing, creating, and deploying a fully-integrated web3 dApp project. Overview dApp projects are divided into two parts: - Frontend: The frontend runs on Next.js. - Smart contracts: The smart contracts use a Hardhat environment. The tool automates basic activity such as auto-updating contract artifacts, (re)deploying contracts on code changes, and Hot Module Replacement (HMR) for frontend code changes, and more. Repo Find the code on the following git repo: https://github.com/0xPolygon/dapp-launchpad !!! warning Please note the docs in the README are possibly not up-to-date. # quickstart.md: Prerequisites - Node version 18.x.x is recommended. - Anything above Node version 16.14.x is supported. - We recommend https://github.com/nvm-sh/nvm for managing Node installations. Install the app Open a terminal window and run the following command to install the launchpad: sh npm install -g @polygonlabs/dapp-launchpad Initialize a new project sh dapp-launchpad init This creates a new directory in your current directory, initializes a minimal dApp project, and installs the required packages. Project templates By default, the scaffolded project uses JavaScript. For TypeScript, use the --template flag. sh dapp-launchpad init --template typescript To see the available templates, run the following: sh dapp-launchpad list scaffold-templates Set up environment variables There are mandatory environment variables in both the frontend and smart-contracts directories. 1. cd into your project directory. 2. Navigate to the frontend folder and copy the example file. sh cd frontend cp .env.example .env 3. Open the .env file and add the NEXTPUBLICWALLETCONNECTPROJECTID to the file. !!! info "How to get the WalletConnect project ID" You can get the variable by creating an application on WalletConnect. 1. Create an account and sign in. 2. Select Create a new project. 3. Give it a name and select Continue. You can leave the URL empty for now. 4. Select AppKit and continue. 5. Select Next.js and then select the Create button the bottom right of the pop-up window. 6. This brings up the dashboard. Copy the project ID form the sidebar on the left and paste it into the frontend .env file. 4. Navigate to the smart-contracts directory and copy the example file. sh cd proj/smart-contracts cp .env.example .env 5. Open the .env file and add the mandatory PRIVATEKEYDEPLOYER variable. This is a private key from any wallet account. !!! info The other variables in the smart-contracts .env file are optional. Start developing 1. Run the following command from the project root: sh dapp-launchpad dev You will see the local test blockchain running with deployed contracts and some pre-funded wallets you can use. !Local test environment running 2. Open http://localhost:3000 in a browser. !Web application running You now have a fully integrated dev environment including a local dev blockchain and a local frontend dev server. Any changes to the code automatically updates both the frontend and the smart contracts. No manual reload is necessary. Start developing on an existing chain fork You can start developing by forking an existing chain. To see the available options run the following: sh dapp-launchpad dev -h The chain name options are in this section of the help output: sh -n, --fork-network-name [NAME] Name of the network to fork; optional. By default, it starts a new chain from genesis block. (choices: "ethereum", "goerli", "polygonPos", "polygonAmoy", "polygonZkevm", "polygonZkevmTestnet") To fork Polygon zkEVM, for example, run the following command: sh dapp-launchpad dev -n polygonZkevm To fork at a particular block number run the command including the optional flag -b: sh dapp-launchpad dev -n polygonZkevm -b [BLOCKNUMBERTOFORKAT] Deploy your app to production To deploy your project to production, run: sh dapp-launchpad deploy -n This does two things: 1. Deploys all your smart contracts to the selected chain, and logs the deployment results. 2. Deploys your frontend to Vercel, and logs the deployment URL. To deploy only the smart contracts, run: sh dapp-launchpad deploy -n CHAINNAME --only-smart-contracts To deploy only the frontend, run: sh dapp-launchpad deploy -n CHAINNAME --only-frontend !!! important The frontend deployment requires that smart contracts to have already been deployed, so if you are only deploying the frontend, make sure that you: 1. Have already run the smart contracts deploy command successfully. 2. If not, run the following wizard command: generate smart-contracts-config -e production -n CHAINNAME # smart-contracts.md: Environment variables Make sure you have followed the steps in the quickstart. Framework The smart contracts run on a Hardhat environment. They are written in Solidity and reside in the smart-contracts directory. Tests are written in JavaScript and TypeScript, and are in the tests directory. An example test is available. Scripts are also written in JavaScript and TypeScript, and reside in the scripts directory. Some mandatory scripts are already there to get started with. Deploying on local test chain Follow the start developing instructions to spin up a local chain. For all available options run: sh dapp-launchpad dev -h Internally, the dev command runs the scripts/deploylocalhost script that deploys all contracts in the correct sequence. !!! warning When working on your own smart contracts, make sure to update this script. Local test chain explorer with Ethernal Optionally, you can enable a local blockchain explorer, which auto-indexes all transactions, and provides a feature-loaded dashboard with an overview of the chain. Prerequisite steps To run the explorer, you first have to: - Sign up on Ethernal, and create a workspace. - Then, add your login email, password, and workspace details in the .env file in the smart-contracts directly. Check and set the configs with dev command params --ethernal-login-email, --ethernal-login-password and --ethernal-workspace, which override the preset environment variables. Run the local block explorer To use it, run the dev command with -e. Access the chain explorer at the Ethernal URL. Deploying to production The deploy command automates deploying to any EVM compatible chain. It runs the provided scripts/deployprod script to deploy all contracts in the correct sequence. When working on your own smart contracts, make sure to update this script. For all available options run: sh dapp-launchpad dev -h # access-node.md: If you're a new Web3 developer, it's unlikely that you'll need to run your own full node on Polygon. The majority of developers use a node provider, or a third-party external service that receives node requests and returns responses for you automatically. That's because the fastest way to get developing on Polygon is using a node provider rather than managing your own node. This guide demonstrates how to connect to a RPC provider, using Alchemy as an example. Send your first blockchain request This guide assumes you already have an Alchemy account and access to the Alchemy Dashboard. Create an Alchemy key First, you'll need an Alchemy API key to authenticate your requests. You can create API keys from the dashboard. Check out this YouTube video on how to create an app. Or you can follow the steps written below: 1. Navigate to the Create App button in the Apps tab. !img 2. Fill in the details under Create App to get your new key. You can also see the applications you previously made by you and your team on this page. Pull existing keys by clicking on View Key for any app. !img !!! tip "Optional" You can also pull existing API keys by hovering over Apps and selecting one. You can View Key here, as well as Edit App to whitelist specific domains, see several developer tools, and view analytics.

!img

Making a cURL request You can interact with Alchemy's Polygon infrastructure provider using JSON-RPC and your command line interface. For manual requests, use POST requests to interact with the JSON-RPC. Simply pass in the Content-Type: application/json header and your query as the POST body with the following fields: jsonrpc: The JSON-RPC versionâ€"currently, only 2.0 is supported. method: The MATIC API method. See API reference. params: A list of parameters to pass to the method. id: The ID of your request. Will be returned by the response so you can keep track of which request a response belongs to. Here is an example you can run from the Terminal/Windows/LINUX command line to retrieve the current gas price: bash curl https://matic-mainnet.alchemyapi.io/v2/demo \ -X POST \ -H "Content-Type: application/json" \ -d '{"jsonrpc":"2.0","method":"ethgasPrice","params":[],"id":73}' !!! info In case you want to send requests to your own app instead of our public demo, replace https://eth-mainnet.alchemyapi.io/jsonrpc/demo with your own API key https://eth-mainnet.alchemyapi.io/v2/your-api-key. Results: json { "id": 73, "jsonrpc": "2.0", "result": "0x09184e72a000" // 10000000000000 } Alchemy SDK setup To make blockchain requests directly from your Javascript / Node.js dApp, you'll need to integrate the Alchemy SDK, the easiest and most powerful way to access the blockchain and Alchemy's suite of enhanced APIs. If you have an existing client such as Web3.js or Ethers.js, you can just change your current node provider URL to an Alchemy URL with your API key: !!! note The scripts below need to be run in a node context or saved in a file, not run from the command line. Install the Alchemy SDK To install the Alchemy SDK, you want to create a project, and then navigate to your project directory to run the installation. Let's go ahead and do that! Once we're in our home directory, let's execute the following: With Yarn: bash mkdir your-project-name cd your-project-name yarn init (or yarn init --yes) yarn add alchemy-sdk With NPM: bash mkdir your-project-name cd your-project-name npm init (or npm init --yes) npm install alchemy-sdk Create index.js Make sure to replace demo with your Alchemy HTTP API key! js title="index.js" // Setup: npm install alchemy-sdk const { Network, Alchemy } = require("alchemy-sdk"); // Optional Config object, but defaults to demo api-key and eth-mainnet. const settings = { apiKey: "demo", // Replace with your Alchemy API Key. network: Network.MATICMAINNET, // Replace with your network. }; const alchemy = new Alchemy(settings); async function main() { const latestBlock = await alchemy.core.getBlockNumber(); console.log("The latest block number is", latestBlock); } main(); If you are unfamiliar with the async stuff, check out this Medium post. Run index.js using node.js bash node index.js Console output You should now see the latest block number output in your console. bash The latest block number is 11043912 Excellent! You just wrote a working Web3 script and sent your first request to your Alchemy API endpoint. The project associated with your API key should now look like this on the dashboard: !img !img Start building Don't know where to start? Check out these tutorials to get more familiar with Alchemy and blockchain development: 1. Examples of Common Queries Using the Alchemy SDK. 3. Learn How to Send Transactions on Ethereum. 4. Try deploying your first Hello World Smart Contract and get your hands dirty with some solidity programming! Other Web3 libraries There are a number of alternative Web3 libraries other than the Alchemy SDK you may already be using. See the documentation for these libraries below: Web3.py Web3j Ethers.js Web3.js Using the below code snippets, you can install and use Alchemy as a provider via any of the following libraries. python Setup: pip install web3 from web3 import Web3 alchemy = Web3(Web3.HTTPProvider("https://eth-mainnet.alchemyapi.io/v2/your-api-key")); js // Setup: curl -L get.web3j.io | sh Web3j web3 = Web3j.build(new HttpService("https://eth-mainnet.alchemyapi.io/v2/your-api-key")); js // Setup: npm install ethers const ethers = require("ethers"); const url = "https://eth-mainnet.alchemyapi.io/v2/your-api-key"; const customHttpProvider = new ethers.providers.JsonRpcProvider(url); js // Setup: npm install web3 const Web3 = require('web3'); const web3 = new Web3("https://eth-mainnet.alchemyapi.io/v2/your-api-key"); # covalent.md: !!! note "Content disclaimer" Please view the third-party content disclaimer here. !!! tip "Quickstart" Check out this introductory video to get started. Supported endpoints All Class A endpoints are supported for the Matic mainnet and the Amoy testnet. You can query either network via the unified API by changing the chainName. !!! info "Endpoints" A full list of all the requests you are able to do on the Polygon network using Covalent are available on the Covalent API documentation. Appendix MATIC gas token To interact with Polygon network, MATIC tokens are required to pay gas fees. Covalent's responses automatically return gas fields in MATIC units. Token mapping Covalent maintains an on-chain real-time mapping of token addresses between the Ethereum mainnet and Polygon. These addresses are used to reverse-lookup prices on MATIC and also to return the right token logo URLs. Some example of mapped tokens: |Token|Ethereum mainnet|Matic mainnet| |---|---|---| |USDT|0xdac17f958d2ee523a2206206994597c13d831ec7|0xc2132d05d31c914a87c6611c10748aeb04b58e8f| |Uniswap UNI|0x1f9840a85d5af5bf1d1762f925bdaddc4201f984|0xb33eaad8d922b1083446dc23f610c2567fb5180f| Token prices For tokens that have a mapping back to Ethereum mainnet, Covalent is able to return the mapped prices. # envio.md: !!! note "Content disclaimer" Please view the third-party content disclaimer here. Envio HyperIndex Envio HyperIndex is a feature-rich indexing solution that provides developers with a seamless and efficient way to index and aggregate real-time or historical blockchain data for any EVM. The indexed data is easily accessible through custom GraphQL queries, providing developers with the flexibility and power to retrieve specific information. Envio offers native support for Polygon PoS (testnet & mainnet), Polygon zkEVM (testnet & mainnet), and Polygon CDK chains, and has been designed to support high-throughput blockchain applications that rely on real-time data for their business requirements. Designed to optimize the user experience, Envio offers automatic code generation, flexible language support, quickstart templates, and a reliable cost-effective hosted service. Indexers on Envio can be written in JavaScript, TypeScript, or ReScript. Envio HyperSync Envio supports HyperSync on Polygon networks. HyperSync is an accelerated data query layer for the Polygon networks, providing APIs that bypass JSON-RPC for 100x faster syncing of historical data. HyperSync is used by default in Envio HyperIndex, with the use of RPC being optional. Using HyperSync, application developers do not need to worry about RPC URLs, rate-limiting, or managing infrastructure and can easily sync large datasets in a few minutes, something that would usually take hours or days via RPC. HyperSync is also available as a standalone API for data analytic use cases. Data analysts can interact with the HyperSync API using JavaScript, Python, or Rust clients and extract data in JSON, Arrow, or Parquet formats. For more information, visit the HyperSync documentation here. HyperIndex key features - Contract import: Autogenerate the key boilerplate for an entire Indexer project off a single or multiple smart contracts. Deploy within minutes. - Multi-chain support: Aggregate data across multiple networks into a single database. Query all your data with a unified GraphQL API. - Asynchronous mode: Fetch data from off-chain storage such as IPFS, or contract state (e.g. smart contract view functions). - Factory contracts: Automatically process events emitted by all child contracts that are created by the specified factory. - Quickstart templates: Use pre-defined indexing logic for popular OpenZeppelin contracts (e.g. ERC-20). Getting started Users can choose whether they want to start from a quickstart template, perform a subgraph migration, or use the contract import feature to get started with Envio HyperIndex. The following files are required to run the Envio indexer: - Configuration (defaults to config.yaml) - GraphQL Schema (defaults to schema.graphql) - Event Handlers (defaults to src/EventHandlers. depending on the language chosen) These files are auto-generated according to the template and language chosen by running the envio init command. Contract import tutorial This walkthrough explains how to initialize an indexer using a single or multiple contracts that are already deployed on Polygon. This process allows a user to quickly and easily start up a basic indexer and a queryable GraphQL API for their application in less than 3 minutes. Initialize your indexer cd into the folder of your choice and run bash envio init Name your indexer bash ? Name your indexer: Choose the directory where you would like to setup your project (default is the current directory) bash ? Set the directory: (.) . Select Contract Import as the initialization option. bash ? Choose an initialization option Template > ContractImport [â†'â†" to move, enter to select, type to filter] bash ? Would you like to import from a block explorer or a local abi? > Block Explorer Local ABI [â†'â†" to move, enter to select, type to filter] Block Explorer option only requires user to input the contracts address and chain of the contract. If the contract is verified and deployed on one of the supported chains, this is the quickest setup as it will retrieve all needed contract information from a block explorer. Local ABI option will allow you to point to a JSON file containing the smart contract ABI. The Contract Import process will then populate the required files from the ABI. Select the blockchain that the contract is deployed on bash ? Which blockchain would you like to import a contract from? ethereum-mainnet arbitrum-one arbitrum-nova > polygon gnosis v optimism [â†'â†" to move, enter to select, type to filter] Enter the address of the contract to import bash ? What is the address of the contract? [Use the proxy address if your abi is a proxy implementation] Note if you are using a proxy contract with an implementation, the address should be for the proxy contract. Choose which events to include in the config.yaml file bash ? Which events would you like to index? > [x] ClaimRewards(address indexed from, address indexed reward, uint256 amount) [x] Deposit(address indexed from, uint256 indexed tokenId, uint256 amount) [x] NotifyReward(address indexed from, address indexed reward, uint256 indexed epoch, uint256 amount) [x] Withdraw(address indexed from, uint256 indexed tokenId, uint256 amount) [â†'â†" to move, space to select one, â†' to all, â† to none, type to filter] Select the continuation option bash ? Would you like to add another contract? > I'm finished Add a new address for same contract on same network Add a new network for same contract Add a new contract (with a different ABI) [Current contract: BribeVotingReward, on network: polygon] The Contract Import process will prompt the user whether they would like to finish the import process or continue adding more addresses for same contract on same network, addresses for same contract on different network or a different contract. For more information on contract import feature, visit the documentation here. Envio examples Click here for Envio HyperIndex examples. Click here for Envio HyperSync examples. Getting help Indexing can be a rollercoaster, especially for more complex use cases. Our engineers are available to help you with your data availability needs. Join our growing community of elite builders, and find peace of mind with Envio. Discord Email: hello@envio.dev # flair.md: !!! note "Content disclaimer" Please view the third-party content disclaimer here. Real-time and historical custom data indexing for any evm chain. Flair offers reusable indexing primitives (such as fault-tolerant RPC ingestors, custom processors, re-org aware database integrations) to make it easy to receive, transform, store and access your on-chain data. !flair architecture Why Flair? Compared to other alternatives the main reasons are: ðŸš€ Adopting parallel and distributed processing paradigm means high scalability and resiliency for your indexing stack. Instead of constrained sequential processing (e.g Subgraph). ðŸ§© Focused on primitives, which means on the left you plug-in an RPC and on the right you output the data to any destination database. ðŸš„ Native real-time stream processing for certain data workload (such as aggregations, rollups) for things like total volume per pool, or total portfolio per user wallet. â˜ï¸ Managed cloud services avoid DevOps and irrelevant engineering costs for dApp developers. ðŸ§'â€ðŸ'» Avoid decentralization overhead (consensus, network hops, etc) since we believe to enable best UX for dApps reading data must be as close to the developers as possible. Features âœ… Listen to any EVM chain with just an RPC URL. Free managed RPC URLs for +8 popular chains already included. Works with both websocket and https-only RPCs. âœ… Track and ingest any contract for any event topic. Auto-track new contracts deployed from factory contracts. âœ… Custom processor scripts with Javascript runtime (with Typescript support) Make external API or Webhook calls to third-party or your backend. Get current or historical USD value of any ERC20 token amount of any contract address on any chain. Use any external NPM library. âœ… Stream any stored data to your destination database (Postgres, MongoDB, MySQL, Kafka, Elasticsearch, Timescale, etc). Getting started 1ï¸âƒ£ Clone the starter boilerplate template and follow the instructions bash git clone https://github.com/flair-sdk/starter-boilerplate.git ... follow instructions in README.md !!! info Boilerplate instructions will create a new cluster, generate an API Key, and set up a manifest.yml to index your first contract with sample custom processor scripts. Learn more about the structure of manifest.yml. 2ï¸âƒ£ Configure Polygon RPC nodes Set a unique namespace, Polygon chainId and RPC endpoint in your config. Remember that you can add up to 10 RPC endpoints for resiliency. yaml { "cluster": "dev", "namespace": "my-awesome-polygon-indexing-dev", "indexers": [{ "chainId": 137, "enabled": true, "ingestionFilterGroup": "default", "processingFilterGroup": "default", "sources": [Highly-recommended to have at least 1 websocket endpoint "wss://rpc-mainnet.matic.network", You can put multiple endpoints for failover "https://polygon-rpc.com/"] }] } 3ï¸âƒ£ Sync some historical data using backfill command. Remember that enabled: true flag in your config enabled your indexer to capture data in real-time already. bash backfill certain contracts or block ranges pnpm flair backfill --chain 137 --address 0x06c9823460b76acf26a9af2acfdcac0d03ca6555 -d backward --max-blocks 10000 backfill for a specific block number, if you have certain events you wanna test with pnpm flair backfill --chain 137 -b 47807770 backfill for the recent data in the last X minute pnpm flair backfill --chain 137 --min-timestamp="30 mins ago" -d backward 4ï¸âƒ£ Query your custom indexed data. 5ï¸âƒ£ Stream the data to your own database. Examples Explore real-world usage of Flair indexing primitives for various use-cases. DeFi Aggregate protocol fees in USD across multiple chains. Calculate "Health Factor" of positions with contract factory tracking. Index Uniswap v2 swaps with USD price for all addresses. NFT Index ERC721 and ERC1155 NFTs on any EVM chain with an RPC URL. Need help? Our engineers are available to help you at any stage. # moralis.md: !!! note "Content disclaimer" Please view the third-party content disclaimer here. About Moralis Moralis is a blockchain data platform that provides developers with all the data they need to build better blockchain applications. From NFT data, token data and price data, through to raw blockchain data and RPC nodes, Moralis offers a wide range of products that cover all major crypto and blockchain use cases, and it supports Polygon together with all other major EVM chains. !!! tip "Quickstart" Check out our popular Youtube channel for many great tutorials on how to get started. Get started Get started with Moralis APIs on Polygon by signup up for a free account. Then you can check out our get started guide. Moralis APIs All Moralis APIs have support for Polygon and across all other major EVM blockchains. All endpoints have powerful filtering capabilities. !!! info "Endpoints" Please check out the Moralis API Reference to see all the available API endpoints. Wallet API With Moralis Wallet API you can get wallet balances for tokens, NFTs and native assets, get full wallet history, net worth and a lot more. NFT API With Moralis NFT API you can get NFT data like collections, owners, prices, images and metadata. Token API With Moralis Token API you can get ERC20 token data like prices, ownership, metadata, transfers, approvals, liquidity, mints and burns. RPC nodes Get access to powerful RPC nodes on all major chains with Moralis nodes. # parsiq.md: !!! note "Content disclaimer" Please view the third-party content disclaimer here. About PARSIQ PARSIQ is a reliable, fully customizable blockchain data indexer, helping developers to seamlessly access, process and utilize Web3 data - both raw and custom. !!! tip "Quickstart" Check out this introduction video to get started. PARSIQ Tsunami API PARSIQ Tsunami API is a highly efficient API to fetch raw Web3 data: - Events, calls, transactions (internal included), transfers, contracts, blocks - you name it. Possibility to use unlimited blockrange makes Tsunami a hard-to-beat solution for realiably getting large amounts of data from the blockchain. CSV Export is available. - Get decoded, human readable data right out of the box. - Need an up to date feeds of data streamed to you in real time? Give our low latency Real Time Streaming service a try. !!! info "Endpoints" Please check out our PARSIQ API Reference to see all the available endpoints. PARSIQ SDK Some more complicated cases where custom data needs to be stored, accumulated, and calculated, cannot be covered by an API. In that cases, use PARSIQ SDK or go for a Custom Data Lake. They allow you to set up data bases and data processing logic to solve your specific use case. !!! info "SDK Documentation" Please see PARSIQ SDK documentation for more details. # sqd.md: !!! note "Content disclaimer" Please view the third-party content disclaimer here. About SQD SQD (previously Subsquid) unlocks next-gen functionality for dApps, AI agents, analytics and ZK Proofs/TEE through blazing-fast indexing. SQD is a decentralized hyper-scalable data platform optimized for providing efficient, permissionless access to large volumes of data. It currently serves historical on-chain data, including event logs, transaction receipts, traces, and per-transaction state diffs. SQD offers a powerful toolkit for creating custom data extraction and processing pipelines, achieving an indexing speed of up to 150k blocks per second. To get started, visit the documentation or see EVM examples of what you can build with SQD. Supported Networks - Polygon Mainnet - Polygon Amoy Testnet - Polygon zkEVM - Polygon zkEVM Cardona Testnet Why SQD? - Faster Development: SQD simplifies blockchain data access with its Squid SDK, allowing developers to write custom queries without getting bogged down in low-level blockchain protocols. This translates to quicker development times. - Cost-Effective: SQD offers free retrieval of historical blockchain data, eliminating a significant cost barrier for developers building data-driven applications. - Scalability: The decentralized architecture of SQD allows for infinite horizontal scaling, meaning it can handle massive amounts of data without compromising performance. - Flexibility: SQD supports a wide range of blockchains (including Layer 1 and Layer 2) and provides a developer-friendly interface for building custom indexers tailored to specific needs. What can you build with SQD? Here is an incomplete list of ideas of what one can build with SQD: - DeFi dashboards, tracking addresses and internal transactions - NFT marketplaces, with a dynamic sets of NFT contracts to watch - Historical price feeds, tracking Uniswap trades and Chainlink oracle contracts - Mining smart contract deployments and the bytecode - Real-time bots (<1sec delay) triggered by on-chain activity SQD also enables cutting-edge use cases like AI agents, RAGs, ZK proofs, and TEE coprocessors. How to host SQD APIs There are generally two ways how you can host your SQD APIs: - Self-Hosting: You can manage your own infrastructure and deploy your SQD project (squid) yourself. - SQD Cloud: This is a hosted service by SQD that takes care of the infrastructure management for you. Getting help Get started faster! Join our Discord community and ask for help. Our engineers are ready to assist you. # subquery.md: !!! note "Content disclaimer" Please view the third-party content disclaimer here. Intro SubQuery is a leading blockchain data indexer that provides developers with fast, flexible, universal, open source and decentralised APIs for web3 projects. SubQuery SDK allows developers to get rich indexed data and build intuitive and immersive decentralised applications in a faster and more efficient way. SubQuery supports 100+ ecosystems including Polygon, Polygon zkEVM, Polygon Amoy, Ethereum, Polkadot, Algorand, NEAR, and Avalanche. Another one of SubQuery's competitive advantages is the ability to aggregate data not only within a chain but across multiple blockchains all within a single project. This allows the creation of feature-rich dashboard analytics and multi-chain block scanners. Other advantages include superior performance with multiple RPC endpoint configurations, multi-worker capabilities and a configurable caching architecture. To find out more, visit our documentation. Useful resources: - SubQuery docs: SubQuery Academy (Documentation) - Intro quick start guide: 1. Create a New Project - Polygon Quick Start Guide - Polygon ZKEVM quick start guide - Polygon Plasma bridge multi-chain quick start guide - Polygon's starter projects directory For technical questions and support reach out to us start@subquery.network Running and hosting your Polygon SubQuery APIs SubQuery is open-source, meaning you have the freedom to run it in the following three ways: - Locally on your own computer (or a cloud provider of your choosing), view the instructions on how to run SubQuery Locally. - You can publish it to SubQuery's enterprise-level Managed Service, where we'll host your SubQuery project in production ready services for mission critical data with zero-downtime blue/green deployments. There even is a generous free tier. Find out how. - You can publish it to the decentralised SubQuery Network, the most open, performant, reliable, and scalable data service for dApp developers. The SubQuery Network indexes and services data to the global community in an incentivised and verifiable way and supports Polygon from launch. # matic-faucet.md: --- comments: true --- !!! info "Testnet tokens hold no real-world value" Tokens on testnets such as PoS Amoy, Ethereum Sepolia, and zkEVM Cardona hold no real value and are solely used for testing purposes. Several faucet tools are available to receive test POL and ETH on Sepolia-anchored Polygon networks: | Faucet | Supported testnets | | :---------------------: | -- | | Polygon Faucet | zkEVM Cardona, PoS Amoy, Ethereum Sepolia | | Alchemy Fauc PoS Amoy \| Ethereum Sepolia | | QuickNode Polygon Faucet | PoS Amoy | | GetBlock Faucet | PoS Amoy \| Ethereum Sepolia | | StakePool Faucet | PoS Amoy, Ethereum Sepolia | Polygon Faucet The Polygon Faucet is a tool provided by Polygon and ecosystem partners like Alchemy, QuickNode, and GetBlock. It enables developers to acquire free POL/ETH test tokens on the PoS Amoy and zkEVM Cardona testnets. Start deploying and testing your dApps on the Polygon network with test tokens that allow you to explore Polygon features without spending actual POL tokens on the mainnet. Polygon Faucet drips: - POL on PoS Amoy. - POL and a dummy ERC20 token on Ethereum Sepolia. - Test ETH on zkEVM Cardona. - Start by navigating to faucet.polygon.technology and connecting/verifying your Discord by selecting Connect Discord in the top-right corner of the page.

!img{width=70%}

- Select one of the blockchain testnet networks from the drop-down list. We'll select Polygon zkEVM (cardona) for this example. - Select the type of token you want to receive; zkEVM ETH in this case. - Copy your wallet address and either paste it directly in the Wallet Address field, or select Paste. Typing is disabled to prevent entering a wrong address. - Select Submit. - Verify the details and select Confirm to finalize the transaction.

!img{width=50%}

- Testnet tokens usually arrive within 1 to 2 minutes.

!img{width=50%}

!!! tip "Always maintain sufficient test token balance" If you don't have enough POL/ETH test tokens in your wallet account to pay for gas fees when sending transactions, the action will fail. If you require testnet tokens in bulk, please fill out this form. Alchemy Faucet !!! tip "Get 0.5 Amoy POL" If you sign up or log in with an Alchemy account, you get 0.5 POL per day. Without an account, you get 0.2. - You can request POL tokens from Alchemy's Amoy Faucet. In order to receive testnet tokens, head over to the Alchemy Polygon Amoy POL faucet, enter your wallet address and hit the Send Me POL button. !img - Check the transaction hash as soon as you finish requesting your POL, and view the transaction details on Polygonscan. QuickNode Polygon Faucet In order to receive Amoy POL tokens using QuickNode faucet, - Head over to the QuickNode Polygon Faucet. Connect your Coinbase, MetaMask, or Phantom wallet by clicking on Connect Wallet. Then, choose Amoy, and finally select Continue.

!img{width=80%}

!!! tip "Get 2x POL" Tweet the drip faucet URL with your wallet address to get twice as many POL testnet tokens. - Click on the transaction hash to view transaction details on Polygonscan. GetBlock Polygon Faucet Head over to the GetBlock Polygon Faucet to acquire Amoy POL tokens using the GetBlock faucet. Please register or login to request free testnet tokens. !!! tip "Get additional tokens" If you tweet about the GetBlock Faucet, you can claim additional PoS Amoy testnet tokens. Discord community Make sure to connect with the Polygon Discord community for news, tips, and help. # polygon-gas-station.md: Polygon gas station aims to help dApp developers with gas price recommendations, so that they can use it before sending transaction off to the Polygon network. We received a lot of requests from dApp developers for building a gas price recommendation service. So we took some inspiration from Eth Gas Station and built one. Polygon gas station has been deployed both on the Polygon mainnet & Amoy testnet. It queries the RPC for ethfeeHistory and fetches the 10th, 25th, and 50th percentiles of priority fees for transactions in each of the last 15 blocks. The average value of the 10th, 25th, and 50th percentiles become the safeLow, standard, and fast fee predictions. !!! important On Polygon PoS mainnet, it is mandatory to pass a minimum priority fees of 30 gwei. Usage Testnet You can send GET requests using the following URLs to fetch gas price recommendations from the gas oracle: - Amoy testnet: https://gasstation.polygon.technology/amoy - zkEVM Cardona: https://gasstation.polygon.technology/zkevm/cardona Use the Amoy testnet below to get gas price recommendations: cURL bash curl https://gasstation.polygon.technology/amoy JavaScript javascript fetch('https://gasstation.polygon.technology/amoy') .then(response => response.json()) .then(json => console.log(json)) Python python import requests requests.get('https://gasstation.polygon.technology/amoy').json() Mainnet You can send GET requests using the following URLs to fetch gas price recommendations from the gas oracle: - PoS mainnet: https://gasstation.polygon.technology/v2 - zkEVM mainnet: https://gasstation.polygon.technology/zkevm cURL bash curl https://gasstation.polygon.technology/v2 JavaScript javascript fetch('https://gasstation.polygon.technology/v2') .then(response => response.json()) .then(json => console.log(json)) Python python import requests requests.get('https://gasstation.polygon.technology/v2').json() Interpretation An example JSON response will look like this: json { "safeLow": { "maxPriorityFee":30.7611840636, "maxFee":30.7611840796 }, "standard": { "maxPriorityFee":32.146027800733336, "maxFee":32.14602781673334 }, "fast": { "maxPriorityFee":33.284344224133335, "maxFee":33.284344240133336 }, "estimatedBaseFee":1.6e-8, "blockTime":6, "blockNumber":24962816 } - safelow, standard, fast, estimatedBaseFee are gas prices in GWei. You can use these prices before sending transaction off to Polygon, depending upon your needs. - blockNumber tells what was latest block mined when recommendation was made. - blockTime, in second, gives average block time of the network. - On Polygon PoS mainnet, it is mandatory to pass a minimum priority fees of 30 gwei. # api-architecture.md: The Matic.js library follows common api architecture throughout and the APIs are divided into two types: 1. Read API 2. Write API Read API Read methods do not publish anything on blockchain, so they do not consume any gas. For example: js const erc20 = posClient.erc20(''); const balance = await erc20.getBalance('') Read methods are very simple and return results directly. Write API Write methods publish some data on the blockchain, so they consumes gas. Example of write methods are approve, deposit etc. A write method returns (at least) two data items: 1. TransactionHash 2. TransactionReceipt For example: js // get a contract object const erc20 = posClient.erc20(''); // send the transaction const result = await erc20.approve(10); // get transaction hash const txHash = await result.getTransactionHash(); // get receipt const receipt = await result.getReceipt(); Transaction option There are some configurable options that are available for all API's. These configurations can be passed as parameters. Available configurations are - - from?: string | number - The address transactions should be made from. - to?: string - The address transactions should be made to. - value?: number | string | BN - The value transferred for the transaction in wei. - gasLimit?: number | string - The maximum gas provided for a transaction (gas limit). - gasPrice?: number | string | BN - The gas price in wei to use for transactions. - data?: string - The byte code of the contract. - nonce?: number; - chainId?: number; - chain?: string; - hardfork?: string; - returnTransaction?: boolean - making it true will return the transaction object which can be used to send transaction manually. This example configures the gas price: js // get a contract object const erc20RootToken = posClient.erc20(,true); // approve 100 amount const approveResult = await erc20Token.approve(100, { gasPrice: '4000000000', }); const txHash = await approveResult.getTransactionHash(); const txReceipt = await approveResult.getReceipt(); # get-started.md: Matic.js a JavaScript library that facilitates interaction with the Polygon network. It provides developers with simplified operations such as depositing, transferring, and withdrawing assets, allowing them to focus on building dApps without requiring in-depth knowledge of blockchain systems. Matic.js supports two popular libraries: 1. Web3.js 2. Ethers Prerequisites Ensure that your Node.js version is up to date. We recommend using Node.js version v18.19.1 for optimal compatibility. Installation Matic.js core library The library contains core logic and provides different APIs. The user interacts mostly with this library. sh npm i @maticnetwork/maticjs Matic.js ethers library The ethers library allows us to use any ethers function. It is injected into Matic.js using plugins. sh npm install @maticnetwork/maticjs-ethers Matic.js web3 library The web3 library allows us to use any web3 function. sh npm install @maticnetwork/maticjs-web3 Initializing To code with matic, import the relevant libraries in your scripts. For example: javascript import { use } from '@maticnetwork/maticjs' import { Web3ClientPlugin } from '@maticnetwork/maticjs-web3' // install web3 plugin use(Web3ClientPlugin) - Click for more details on POS applications that use web3js. - Click for more details on POS applications that use ethers. Support In case you face any issues or have any queries, feel free to raise a ticket to our Support Team or reach out to us on Discord. # set-proof-api.md: !!! note The API links have been changed from https://apis.matic.network → https://proof-generator.polygon.technology. Please make sure to update the links. Some of the functions in the MaticJS library are suffixed with the term faster. As the name suggests, they generate results more quickly compared to their non-faster counterparts. They do so by utilizing the Proof Generation API at the backend which anyone can host. Proof Generator is a publicly available proof generation API, hosted by Polygon Labs. The setProofApi method can help in setting the Proof Generation APIâ€™s URL to the MaticJS instance as demonstrated in the below snippet. js import { setProofApi } from '@maticnetwork/maticjs' setProofApi("https://proof-generator.polygon.technology/"); !!! info Utilizing a self-hosted Proof Generation API service will offer better performance compared to a publicly hosted one. Please follow the installation instructions provided in the README.md file in this repository to self-host the service. Let's say you have deployed the Proof API and your base URL is https://abc.xyz/. In this case, you need to set the base URL in setProofApi as follows. js import { setProofApi } from '@maticnetwork/maticjs' setProofApi("https://abc.xyz/"); We recommend using faster APIs because some APIs, particularly where proof is being generated, make a lot of RPC calls and it might be very slow with public RPCs. # abi-manager.md: matic.js internally uses ABIManager for handling ABI management, configuration for you. All of the ABI and config are taken from static repo. Change ABI Sometimes you are required to change the ABI, particularly when you are developing a contract. You can do so by using ABIManager. Syntax js import { ABIManager } from '@maticnetwork/maticjs' const manager = new ABIManager(,); await manager.init(); // set abi manager.setABI(,,); // get abi manager.getABI(,); The network name, contract name, bridge name etc can be taken from our official static repo. Example js import { ABIManager } from '@maticnetwork/maticjs' const manager = new ABIManager('testnet','amoy'); await manager.init(); // set abi manager.setABI('ERC20PredicateProxy','pos', 'abi value'); // get abi manager.getABI('ERC20PredicateProxy','pos'); # exit-util.md: matic.js internally use ExitUtil for generating proofs. It is a class which has different methods for helping with exit utilities. buildPayloadForExit It exposes buildPayloadForExit method which can be used to generate proof. js import { ExitUtil, RootChain, use, Web3SideChainClient } from "@maticnetwork/maticjs"; import { Web3ClientPlugin } from "@maticnetwork/maticjs-web3"; import HDWalletProvider from "@truffle/hdwallet-provider"; import { from, privateKey, RPC } from "./config"; use(Web3ClientPlugin); const client = new Web3SideChainClient(); // initiate client await client.init({ // log: true, network: 'testnet', version: 'amoy', parent: { provider: new HDWalletProvider(privateKey, RPC.parent), defaultConfig: { from } }, child: { provider: new HDWalletProvider(privateKey, RPC.child), defaultConfig: { from } } }); // create root chain instance const rootChain = new RootChain(client,); // create exitUtil Instance const exitUtil = new ExitUtil(client, rootChain); // generate proof const proof = await exitUtil.buildPayloadForExit(, ,) Generating proof using a bridge client Every bridge client, including POSClient exposes the exitUtil property. js import { POSClient,use } from "@maticnetwork/maticjs" import { Web3ClientPlugin } from '@maticnetwork/maticjs-web3' import HDWalletProvider from "@truffle/hdwallet-provider" // install web3 plugin use(Web3ClientPlugin); const posClient = new POSClient(); await posClient.init({ network: 'testnet', version: 'amoy', parent: { provider: new HDWalletProvider(privateKey, mainRPC), defaultConfig: { from : fromAddress } }, child: { provider: new HDWalletProvider(privateKey, childRPC), defaultConfig: { from : fromAddress } } }); const proof = await posClient.exitUtil.buildPayloadForExit(, ,) # plugin.md: Using plugin you can inject your code into matic.js. It can be used to write common set of generic codes which can be provided to anyone using a package. !!! info Plugin makes the matic.js light weight as it implements only important logical part. In fact, the web3 library is supported using plugin which allows us to use our favorite library. Plugin development Plugin is a class which implements IPlugin. js import { IPlugin } from "@maticnetwork/maticjs"; export class MyPlugin implements IPlugin { // variable matic is - default export of matic.js setup(matic) { // get web3client const web3Client = matic.Web3Client ; } } As you can see - you just need to implement a setup method which will be called with default export of matic.js. Use plugin matic.js expose use method for using a plugin. js import { use } from '@maticnetwork/maticjs' use(MyPlugin) You can use multiple plugins and they will be called in the same order as they are declared. Some plugin repos are - - Matic web3.js - Matic ethers - FxPortal.js # client.md: !!! important Make sure you have set up Matic.js by following the get started guide. The POSClient interacts with the POS bridge. js import { POSClient,use } from "@maticnetwork/maticjs" import { Web3ClientPlugin } from '@maticnetwork/maticjs-web3' import HDWalletProvider from "@truffle/hdwallet-provider" // install web3 plugin use(Web3ClientPlugin); const posClient = new POSClient(); await posClient.init({ network: 'testnet', version: 'amoy', parent: { provider: new HDWalletProvider(privateKey, mainRPC), defaultConfig: { from : fromAddress } }, child: { provider: new HDWalletProvider(privateKey, childRPC), defaultConfig: { from : fromAddress } } }); After the POSClient is initiated, we can work with token types, such as erc20, erc721 etc. ERC20 Create ERC20 child token js const erc20ChildToken = posClient.erc20(); Create ERC20 parent token js const erc20ParentToken

= posClient.erc20(, true); Once erc20 is initaited, you can call various methods that are available, like - getBalance, approve, deposit , withdraw etc. getBalance js const balance = await erc20ChildToken.getBalance() console.log('balance', balance) approve js // approve amount 10 on parent token const approveResult = await erc20ParentToken.approve(10); // get transaction hash const txHash = await approveResult.getTransactionHash(); // get transaction receipt const txReceipt = await approveResult.getReceipt(); As you can see, with its simple APIs maticjs makes it very easy to interact with maticjs bridge. Useful links - Examples Once the POSClient is initiated, you can interact with all available APIs. # deposit-ether.md: Use the depositEther method to deposit ETH from ethereum to polygon. For example: js const result = await posClient.depositEther(,); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); # is-check-pointed.md: The isCheckPointed method can be used to know if a transaction has been check-pointed. js const isCheckPointed = await posClient.isCheckPointed(); # is-deposited.md: The isDeposited method shows if a deposit has been completed. js const isDeposited = await posClient.isDeposited(); # approve-all-for-mintable.md: The approveAllForMintable method can be used to approve all mintable tokens on root token. js const erc115RootToken = posClient.erc115(,true); const approveResult = await erc115RootToken.approveAllForMintable(); const txHash = await approveResult.getTransactionHash(); const txReceipt = await approveResult.getReceipt(); # approve-all.md: The approveAll method can be used to approve all tokens on root token. js const erc1155RootToken = posClient.erc1155(,true); const approveResult = await erc1155RootToken.approveAll(); const txHash = await approveResult.getTransactionHash(); const txReceipt = await approveResult.getReceipt(); # deposit-many.md: The depositMany method can be used to deposit required amounts of multiple token from ethereum to polygon chain. js const erc1155RootToken = posClient.erc1155(, true); const result = await erc1155RootToken.depositMany({ amount: [1,2], tokenId: ['123','456'], userAddress: , data: '0x5465737445524331313535', // data is optional }); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); Supplying data is optional. # deposit.md: The deposit method can be used to deposit required amount of a token from ethereum to polygon chain. js const erc1155RootToken = posClient.erc1155(, true); const result = await erc1155RootToken.deposit({ amount: 1, tokenId: '123', userAddress: , data: '0x5465737445524331313535', // data is optional }); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); Supplying data is optional. # get-balance.md: The getBalance method can be used to get the balance of user for a token. It is available on both child and parent token. js const erc1155Token = posClient.erc1155(); // get balance of user const balance = await erc1155Token.getBalance(,); # index.md: ERC1155 POSClient provides erc1155 method which helps you to interact with a erc1155 token. The method returns instance of ERC1155 class which contains different methods. js const erc721token = posClient.erc1155(,); Passing second arguments for isRoot is optional. Child token The token on Polygon can be initiated by using this syntax - js const childERC20Token = posClient.erc1155(); Parent token The token on Ethereum can be initiated by providing second parameter value as true. js const parentERC20Token = posClient.erc1155(, true); # is-approved-all.md: The isApprovedAll method checks if all tokens are approved for a user. It returns boolean value. js const erc1155Token = posClient.erc1155(, true); const result = await erc1155Token.isApprovedAll(); # is-withdraw-exited-many.md: The isWithdrawExitedMany method check if withdraw has been exited for multiple tokens. It returns boolean value. js const erc1155Token = posClient.erc1155(); const result = await erc1155Token.isWithdrawExitedMany(); # is-withdraw-exited.md: The isWithdrawExited method check if a withdraw has been exited. It returns boolean value. js const erc1155Token = posClient.erc1155(); const result = await erc1155Token.isWithdrawExited(); # transfer.md: The transfer method can be used to transfer tokens from one user to another user. js const erc1155Token = posClient.erc1155(); const result = await erc1155Token.transfer({ tokenId: , amount: , from : , to : , data : , // data is optional }); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); # withdraw-exit-faster-many.md: The withdrawExitFasterMany method can be used to exit the withdraw process by using the txHash from withdrawStartMany method. It is fast because it generates proof in backend. You need to configure setProofAPI. Note- withdrawStart transaction must be checkpointed in order to exit the withdraw. js const erc1155RootToken = posClient.erc1155(, true); const result = await erc1155RootToken.withdrawExitFasterMany(); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); # withdraw-exit-faster.md: The withdrawExitFaster method can be used to exit the withdraw process by using the txHash from withdrawStart method. It is fast because it generates proof in backend. You need to configure setProofAPI. Note- withdrawStart transaction must be check-pointed in order to exit the withdraw. js const erc1155RootToken = posClient.erc1155(, true); const result = await erc1155RootToken.withdrawExitFaster(); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); # withdraw-exit-many.md: The withdrawExitMany method can be used to exit the withdraw process by using the txHash from withdrawStartMany method. js const erc1155RootToken = posClient.erc1155(, true); const result = await erc1155RootToken.withdrawExitMany(); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); # withdraw-exit.md: The withdrawExit method can be used to exit the withdraw process by using the txHash from withdrawStart method. js const erc1155RootToken = posClient.erc1155(, true); const result = await erc1155RootToken.withdrawExit(); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); # withdraw-start-many.md: The withdrawStartMany method can be used to initiate the withdraw process which will burn the specified amounts of multiple token respectively on polygon chain. js const erc1155Token = posClient.erc1155(); const result = await erc1155Token.withdrawStartMany([,],[,]); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); # withdraw-start.md: The withdrawStart method can be used to initiate the withdraw process which will burn the specified amount of tokenId on polygon chain. js const erc1155Token = posClient.erc1155(); const result = await erc1155Token.withdrawStart(,); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); # approve-max.md: The approveMax method can be used to approve max amount on the root token. js const erc20RootToken = posClient.erc20(, true); const approveResult = await erc20RootToken.approveMax(); const txHash = await approveResult.getTransactionHash(); const txReceipt = await approveResult.getReceipt(); spenderAddress The address on which approval is given is called spenderAddress. It is a third-party user or a smart contract which can transfer your token on your behalf. By default spenderAddress value is erc20 predicate address. You can specify spender address value manually. js const erc20RootToken = posClient.erc20(, true); // approve 100 amount const approveResult = await erc20Token.approveMax({ spenderAddress: }); const txHash = await approveResult.getTransactionHash(); const txReceipt = await approveResult.getReceipt(); # approve.md: The approve method can be used to approve required amount on the root token. approve is required in order to deposit amount on polygon chain. js const erc20RootToken = posClient.erc20(,true); // approve 100 amount const approveResult = await erc20Token.approve(100); const txHash = await approveResult.getTransactionHash(); const txReceipt = await approveResult.getReceipt(); spenderAddress The address on which approval is given is called spenderAddress. It is a third-party user or a smart contract which can transfer your token on your behalf. By default spenderAddress value is erc20 predicate address. You can specify spender address value manually. js const erc20RootToken = posClient.erc20(,true); // approve 100 amount const approveResult = await erc20Token.approve(100, { spenderAddress: }); const txHash = await approveResult.getTransactionHash(); const txReceipt = await approveResult.getReceipt(); # deposit.md: The deposit method can be used to deposit required amount from root token to child token. js const erc20RootToken = posClient.erc20(, true); //deposit 100 to user address const result = await erc20Token.deposit(100,); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); It might take some time to see the deposited amount on polygon chain. You can use isDeposited method for checking status. # get-allowance.md: The getAllowance method can be used to get the approved amount for the user. js const erc20Token = posClient.erc20(, true); const balance = await erc20Token.getAllowance(); spenderAddress The address on which approval is given is called spenderAddress. It is a third-party user or a smart contract which can transfer your token on your behalf. By default spenderAddress value is erc20 predicate address. You can specify spender address value manually. js const erc20Token = posClient.erc20(, true); const balance = await erc20Token.getAllowance(, { spenderAddress: }); # get-balance.md: The getBalance method can be used to get the balance of user. It is available on both child and parent token. js const erc20Token = posClient.erc20(); // get balance of user const balance = await erc20Token.getBalance(); # index.md: ERC20 The POSClient has an erc20 method which returns an object of an ERC20 token. You can then call various methods on the object. js const erc20token = posClient.erc20(,); Passing second arguments for isRoot is optional. Child token Token on polygon can be initiated by using this syntax - js const childERC20Token = posClient.erc20(); Parent token Token on ethereum can be initiated by providing the second parameter value as true. js const parentERC20Token = posClient.erc20(, true); # is-withdraw-exited.md: The isWithdrawExited method can be used to know whether the withdraw has been exited or not. js const erc20RootToken = posClient.erc20(,true); const isExited = await erc20Token.isWithdrawExited(); # transfer.md: The transfer method can be used to transfer amount from one address to another address. js const erc20Token = posClient.erc20(); const result = await erc20Token.transfer(,); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); # withdraw-exit-faster.md: The withdrawExitFaster method can be used to exit the withdraw process faster by using the txHash from withdrawStart method. It is generally fast because it generates proof in the backend. You need to configure setProofAPI. Note- withdrawStart transaction must be checkpointed in order to exit the withdraw. js import { setProofApi } from '@maticnetwork/maticjs' setProofApi("https://proof-generator.polygon.technology/"); const erc20RootToken = posClient.erc20(, true); // start withdraw process for 100 amount const result = await erc20Token.withdrawExitFaster(); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); Once the transaction is complete & checkpoint is completed, amount will be deposited to root chain. # withdraw-exit.md: The withdrawExit method can be used to exit the withdraw process by using the txHash from withdrawStart method. Note- withdrawStart transaction must be checkpointed in order to exit the withdraw. js const erc20RootToken = posClient.erc20(, true); const result = await erc20Token.withdrawExit(); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); This method does multiple RPC calls to generate the proof and process exit. So it is recommended to use withdrawExitFaster method. # withdraw-start.md: The withdrawStart method can be used to initiate the withdraw process which will burn the specified amount on polygon chain. js const erc20Token = posClient.erc20(); // start withdraw process for 100 amount const result = await erc20Token.withdrawStart(100); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); The received transaction hash will be used to exit the withdraw process. So we recommend to store it. # approve-all.md: The approveAll method can be used to approve all tokens. js const erc721RootToken = posClient.erc721(, true); const approveResult = await erc721RootToken.approveAll(); const txHash = await approveResult.getTransactionHash(); const txReceipt = await approveResult.getReceipt(); # approve.md: The approve method can be used to approve required amount on root token. js const erc721RootToken = posClient.erc721(,true); const approveResult = await erc721RootToken.approve(); const txHash = await approveResult.getTransactionHash(); const txReceipt = await approveResult.getReceipt(); # deposit-many.md: The depositMany method can be used to deposit multiple token from ethereum to polygon chain. js const erc721RootToken = posClient.erc721(, true); const result = await erc721RootToken.depositMany([,],); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); # deposit.md: The deposit method can be used to deposit a token from ethereum to polygon chain. js const erc721RootToken = posClient.erc721(, true); const result = await erc721RootToken.deposit(,); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); # get-all-tokens.md: The getAllTokens method returns all tokens owned by specified user. js const erc721Token = posClient.erc721(); const result = await erc721RootToken.getAllTokens(,); You can also limit the tokens by specifying the limit value in the second parameter. # get-token-id-at-index-for-user.md: The getTokenIdAtIndexForUser method returns token id on supplied index for user. js const erc721Token = posClient.erc721(); const result = await erc721Token.getTokenIdAtIndexForUser(,); # get-tokens-count.md: The getTokensCount method returns tokens count for specified user. js const erc721Token = posClient.erc721(); const result = await erc721Token.getTokensCount(); # index.md: ERC721 POSClient provides erc721 method which helps you to interact with a erc721 token. The method returns an object which has various methods. const erc721token = posClient.erc721(,); Passing second arguments for isRoot is optional. Child token Token on polygon can be initiated by using this syntax - const childERC20Token = posClient.erc721(); Parent token Token on ethereum can be initiated by providing second parameter value as true. const parentERC20Token = posClient.erc721(, true); # is-approved-all.md: The isApprovedAll method checks if all token is approved. It returns boolean value. js const erc721Token = posClient.erc721(, true); const result = await erc721Token.isApprovedAll(); # is-approved.md: The isApproved method checks if token is approved for specified tokenId. It returns boolean value. js const erc721Token = posClient.erc721(, true); const result = await erc721Token.isApproved(); # is-withdraw-exited-many.md: The isWithdrawExitedMany method check if withdraw has been exited for multiple tokens. It returns boolean value. js const erc721Token = posClient.erc721(); const result = await erc721Token.isWithdrawExitedMany(); # is-withdraw-exited.md: The isWithdrawExited method check if a withdraw has been exited. It returns boolean value. js const erc721Token = posClient.erc721(); const result = await erc721Token.isWithdrawExited(); # transfer.md: The transfer method can be used to transfer tokens from one user to another user. js const erc721Token = posClient.erc721(); const result = await erc721Token.transfer(,,); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); # withdraw-exit-faster-many.md: The withdrawExitFasterMany method can be used to exit the withdraw process by using the txHash from withdrawStartMany method. It is fast because it generates proof in the back-end. You need to configure setProofAPI. Note- withdrawStart transaction must be checkpointed in order to exit the withdraw. js const erc721RootToken = posClient.erc721(, true); const result = await erc721RootToken.withdrawExitFasterMany(); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); # withdraw-exit-faster.md: The withdrawExitFaster method can be used to exit the withdraw process by using the txHash from withdrawStart method. It is fast because it generates proof in the back-end. You need to configure setProofAPI. Note- withdrawStart transaction must be checkpointed in order to exit the withdraw. js const erc721RootToken = posClient.erc721(, true); const result = await erc721RootToken.withdrawExitFaster(); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); # withdraw-exit-many.md: The withdrawExitMany method can be used to exit the withdraw process by using the txHash from withdrawStartMany method. js const erc721RootToken = posClient.erc721(, true); const result = await erc721RootToken.withdrawExitMany(); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); # withdraw-exit.md: withdrawExit method can be used to exit the withdraw process by using the txHash from withdrawStart method. js const erc721RootToken = posClient.erc721(, true); const result = await erc721RootToken.withdrawExit(); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); This method does multiple RPC calls to generate the proof and process exit. So it is recommended to use withdrawExitFaster method. # withdraw-start-many.md: The withdrawStartMany method can be used to initiate the withdraw process which will burn the multiple token on polygon chain. js const erc721Token = posClient.erc721(); const result = await erc721Token.withdrawStartMany([,]); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); # withdraw-start-with-meta-data.md: The withdrawStartWithMetaData method can be used to initiate the withdraw process which will burn the specified token on polygon chain. Under the hood it calls withdrawWithMetadata method on token contract. js const erc721Token = posClient.erc721(); const result = await erc721Token.withdrawStartWithMetaData(); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); # withdraw-start.md: The withdrawStart method can be used to initiate the withdraw process which will burn the specified token on polygon chain. js const erc721Token = posClient.erc721(); const result = await erc721Token.withdrawStart(); const txHash = await result.getTransactionHash(); const txReceipt = await result.getReceipt(); # ethers.md: ethers.js is a library for interacting with the Ethereum Blockchain and its ecosystem. Setup ether.js ether.js support is available via separate package as a plugin for matic.js. Installation sh npm install @maticnetwork/maticjs-ethers Setup js import { use } from '@maticnetwork/maticjs' import { Web3ClientPlugin } from '@maticnetwork/maticjs-ethers' // install ethers plugin use(Web3ClientPlugin) Let's see one example of creating POSClient using ethers - js import { POSClient,use } from "@maticnetwork/maticjs" import { Web3ClientPlugin } from '@maticnetwork/maticjs-ethers' import { providers, Wallet } from "ethers"; // install web3 plugin use(Web3ClientPlugin); const parentProvider = new providers.JsonRpcProvider(rpc.parent); const childProvider = new providers.JsonRpcProvider(rpc.child); const posClient = new POSClient(); await posClient.init({ network: 'testnet', version: 'amoy', parent: { provider: new Wallet(privateKey, parentProvider), defaultConfig: { from : fromAddress } }, child: { provider: new Wallet(privateKey, childProvider), defaultConfig: { from : fromAddress } } }); Examples The examples for different cases are available on ethers plugin repo. # web3js.md: Web3.js is a collection of libraries that allow you to interact with a local or remote Ethereum node using HTTP, IPC or WebSocket. Setup web3.js web3.js support is available via separate package as a plugin for matic.js. Installation sh npm install @maticnetwork/maticjs-web3 Setup js import { use } from '@maticnetwork/maticjs' import { Web3ClientPlugin } from '@maticnetwork/maticjs-web3' // install web3 plugin use(Web3ClientPlugin) Let's see an example of creating POSClient using web3 - js import { POSClient,use } from "@maticnetwork/maticjs" import { Web3ClientPlugin } from '@maticnetwork/maticjs-web3' import HDWalletProvider from "@truffle/hdwallet-provider" // install web3 plugin use(Web3ClientPlugin); const posClient = new POSClient(); await posClient.init({ network: 'testnet', version: 'amoy', parent: { provider: new HDWalletProvider(privateKey, mainRPC), defaultConfig: { from : fromAddress } }, child: { provider: new HDWalletProvider(privateKey, childRPC), defaultConfig: { from : fromAddress } } }); Examples The examples for different cases are available in web3 plugin repo. # common-methods.md: isDeposited The isDeposited method can be used to check if a deposit has been completed. js const isDeposited = await zkEvmClient.isDeposited(txhash); isDepositClaimable The isDepositClaimable method checks if a deposit can be claimed on the network. js const isDepositClaimable = await zkEvmClient.isDepositClaimable(txhash); isWithdrawExitable This method checks if the withdrawal process can be exited. js const isWithdrawExitable = await zkEvmClient.isWithdrawExitable(txhash); isExited isExited method checks if a withdrawal has been exited. It returns a boolean value. js const isExited = await zkEvmClient.isExited(txhash); # erc20.md: Calling the ERC20 method ZkEvmClient provides erc20 method which helps you to interact with an ERC20 token on the zkEVM network. The method returns an object which has various other methods. js const erc20token = zkEvmClient.erc20(,); Passing second argument for isRoot is optional. For child token Token on the zkEVM network can be initiated by using this syntax: js const childERC20Token = zkEvmClient.erc20(); For root token Token on ethereum can be initiated by providing the second parameter value as true. js const rootERC20Token = zkEvmClient.erc20(, true); Check balance You can use the getBalance method to get the balance of a user account. It is available for both child and root token. js // get balance of user const balance = await erc20Token.getBalance(); Approve methods approve The approve method can approve the required amount on the root and child token. It is needed for both deposit and withdrawal on the zkEVM network. Some tokens require this method during withdrawal while others require it during deposit. js const erc20Token = zkEvmClient.erc20(, true); // root token // approve 1000 amount const result = await erc20Token.approve(1000); const txHash = await result.getTransactionHash(); const receipt = await result.getReceipt(); spenderAddress The address on which approval is given is called the spenderAddress. It is a third-party user or a smart contract which can transfer your token on your behalf. By default, spenderAddress value is the PolygonZkEVMBridge contract address. You can specify spenderAddress value manually. js // approve 1000 amount const result = await erc20Token.approve(1000, { spenderAddress: }); approveMax The approveMax method can approve the maximum amount on the root and child tokens. js const erc20Token = zkEvmClient.erc20(, true); // root token const result = await erc20Token.approveMax(); const txHash = await result.getTransactionHash(); const receipt = await result.getReceipt(); spenderAddress You can specify spenderAddress value manually. js // approve 100 amount const result = await erc20Token.approveMax({ spenderAddress: }); isApprovalNeeded isApprovalNeeded checks if approval is needed for the root or child token. js const erc20Token = zkEvmClient.erc20(, true); // root token const result = await erc20Token.isApprovalNeeded(); getAllowance getAllowance method can be used to get the approved amount for the user. js const erc20Token = zkEvmClient.erc20(, true); // root token const result = await erc20Token.getAllowance(); spenderAddress You can specify spender address value manually. js const result = await erc20Token.getAllowance(, { spenderAddress: }); Transfer method The transfer method can be used to transfer amount from one address to another. js const erc20Token = zkEvmClient.erc20(); const result = await erc20Token.transfer(,); const txHash = await result.getTransactionHash(); const receipt = await result.getReceipt(); Deposit methods deposit deposit method can be used to deposit the required amount from root chain to the child chain. We recommend users to store the transaction hash in order to be able to call depositClaim using that txHash. js const erc20Token = zkEvmClient.erc20(, true); // root token //deposit 100 to user address const result = await erc20Token.deposit(100,); const txHash = await result.getTransactionHash(); const receipt = await result.getReceipt(); depositEther depositEther method can be used to deposit required amount of ether from Ethereum to zkEVM. js // ether address = 0x0000000000000000000000000000000000000000 const etherToken = zkEvmClient.erc20(, true); const result = await etherToken.deposit(,); const txHash = await result.getTransactionHash(); const receipt = await result.getReceipt(); depositWithPermit depositWithPermit method can be used to deposit required amount of tokens from Ethereum to zkEVM along with the permit, so that user doesn't have to do multiple transactions for approve and deposit. js const erc20Token = zkEvmClient.erc20(, true); // root token const result = await erc20Token.depositWithPermit(,); const txHash = await result.getTransactionHash(); const receipt = await result.getReceipt(); depositClaim depositClaim method is used for child tokens to claim their ERC20 token deposits. js const erc20Token = zkEvmClient.erc20(); // child token const result = await erc20Token.depositClaim(); // child token const result = await erc20Token.withdraw(, , true); // root token const result = await erc20Token.withdrawExit(); const txHash = await result.getTransactionHash(); const receipt = await result.getReceipt(); # initialize.md: !!! important Make sure you have set up Matic.js by following the get started guide. The ZkEvmClient interacts with the zkEVM bridge. js import { ZkEvmClient, use } from "@maticnetwork/maticjs" const zkEvmClient = new ZkEvmClient(); await zkEvmClient.init({ network: , // 'testnet' version: , // 'blueberry' parent: { provider: , defaultConfig: { from: } }, child: { provider: , defaultConfig: { from: } } }); Once the ZkEvmClient is initialized, you can interact with all available APIs from MaticJS SDK. # message-passing.md: This document demonstrates inter-layer message passing using the messaging layer of the Polygon bridge. As an example, we go over how to customize wrapped tokens using adapter contracts, and how to use Matic.js to bridge assets from Ethereum to Polygon zkEVM and vice versa. !!! info "Terminology" Within the scope of this doc, we refer to Ethereum as the root chain and zkEVM as the child chain. The existing zkEVM bridge uses the ERC20 standard contract for creating wrapped tokens depending on the token's native network. Often, organizations want to customize their wrapped tokens by extending some functionalities. These functionalities could include: blacklisting, putting a cap on minting, or any sound auxiliary functionality. This can be done by deploying adapter contracts that use the messaging layer of the bridge. Adapter contracts An adapter is a wrapper contract that implements the Polygon bridge library. An example implementation for ERC20 can be found here. Ideally, the following adapter contracts are expected, 1. OriginChainBridgeAdapter 2. WrapperChainBridgeAdapter Irrespective of whether an ERC20 token is Ethereum native (root chain) or zkEVM native (child chain), an adapter contract should have the following functions: (these are already part of the library). solidity function bridgeToken(address destinationAddress, uint256 amount, bool forceUpdateGlobalExitRoot) external {} function onMessageReceived(address originAddress, uint32 originNetwork, bytes memory data) external payable {} For the sake of maintaining consistency among wrapped tokens in terms of the bridging mechanism, there are certain standard functions and variables that need to be included in the adapter contracts. Standardizations 1. Adapter contracts need to implement the Polygon bridge library and expose bridgeToken() and onMessageReceived() functions. 2. There should be two separate adapter contracts; OriginChainBridgeAdapter and WrapperChainBridgeAdapter. 3. bridgeToken() function should match the exact function signature and be similar to this ABI. Nice to have Expose the following variables, 1. originTokenAddress: Address of the native token. 2. originTokenNetwork: networkId of the chain to which the token is native. 3. wrappedTokenAddress: Address of the wrapped token. Bridging mechanism PolygonZkEVMBridge is the main bridge contract. It exposes a bridgeMessage() function, which users can call in order to bridge messages from L1 to L2, or vice versa. The claimMessage() function can be called on the receiving chain to claim the sent message. For example, a user who wants to bridge a message from Ethereum to zkEVM, can call bridgeMessage() on Ethereum and then call claimMessage() on zkEVM. Once the claimMessage() function is called, the bridge calls onMessageReceived for the specified destination address. Adapter contracts are basically abstractions that use bridgeMessage() to bridge and onMessageReceived() to process a claimMessage() on respective chains. !Figure: Adapter contract ERC20 transfer contract interaction The transfer of ERC20 tokens using each of the adapter contracts and the actions performed in the process are described below. OriginChainBridgeAdapter When depositing an ERC20 token from Ethereum to zkEVM, the adapter contract calls the bridgeToken() function. During withdrawal from zkEVM, the PolygonZkEvmBridge.sol contract calls the onMessageReceived() function when claimMessage() is invoked. WrapperChainBridgeAdapter When withdrawing an ERC20 token from zkEVM to Ethereum, the adapter contract calls the bridgeToken() function. During a deposit to zkEVM, the PolygonZkEvmBridge.sol contract calls the onMessageReceived() function when claimMessage() is invoked. From Ethereum â†' zkEVM !!! warning It is assumed that the token being bridged is native to the root chain. 1. Deploy your adapter contracts on both the root chain and the child chain. (Note the address, youâ€™ll need it later!) 2. Approve the tokens to be transferred by calling the approve() function (on the root token) with the address of the originChainBridgeAdapter and the token amount, as arguments. 3. Proceed to call bridgeToken() while using as arguments: the recipient, amount, and setting forceUpdateGlobalExitRoot to true on the originChainBridgeAdapter in the root chain (i.e., Ethereum). 4. Get the Merkle proof for this bridge transaction using the proof API. 5. Proceed to call claimMessage() with the respective arguments on the PolygonZkEVMBridge.sol contract in the child chain (i.e., zkEVM). The bridge will call the onMessageReceived function in the WrapperChainBridgeAdapter contract. Which should ideally have the logic to mint wrapped tokens to the recipient. From zkEVM â†' Ethereum !!! warning It is assumed that the token being bridged is native to the root chain. 1. Deploy your adapter contracts on both the root chain and the child chain. (Note the address, youâ€™ll need it later!) 2. Approve the tokens to be transferred by calling the approve() function (on the wrapped token) with the address of the wrapperChainBridgeAdapter and the token amount as arguments. 3. Proceed to call bridgeToken(), using as arguments: the recipient, amount, and setting forceUpdateGlobalExitRoot to true on the WrapperChainBridgeAdapter in the child chain (i.e., zkEVM). Ideally, this function should have the logic to burn the wrapped tokens. 4. Get the Merkle proof for this bridge transaction using the proof API. 5. Proceed to call claimMessage() with the respective arguments on the PolygonZkEVMBridge.sol contract in the root chain (i.e., Ethereum). The bridge will call the onMessageReceived function in the OriginChainBridgeAdapter contract. Which should Ideally have the logic to mint unwrapped tokens to the recipient. Listing tokens in Bridge UI !!! tip Note that it is important to follow standardizations for easy listing. 1. Add your token to this token list on GitHub. Example: solidity { "chainId": 1101, "name": "Token Name", "symbol": "Token Symbol", "decimals": 6, // token decimal "address": "ZkEVM Address of the token", "logoURI": "Token logo url", "tags": ["zkevm", "stablecoin", "erc20, custom-zkevm-bridge"], "originTokenNetwork": 0, // 0 here is networkId of ethereum, "wrappedTokenNetwork": 1, // 1 here is networkId of zkEvm, "extensions": { "rootAddress": "Ethereum Address of the Token", "wrapperChainBridgeAdapter": "", "originChainBridgeAdapter": "", } }, !!! note "Setting the correct network ID" If the token is Ethereum native, then originTokenNetwork should be 0. If the token is zkEVM native, then originTokenNetwork should be 1. The same rule applies for the wrappedTokenNetwork field. 2. Raise a PR ðŸš€. Using Matic.js to bridge using adapter contracts Deploy your OriginChainBridgeAdapt and WrapperChainBridgeAdapter. Make sure you are using matic.js version > 3.6.4. - Create an instance of the zkEVM client, passing the necessary parameters. Refer here for more info. jsx const client = new ZkEvmClient(); await client.init({}) - Create an ERC20 token instance which you would like to bridge, jsx const erc20Token = client.erc20("", "", ""); Bridge from Ethereum â†' zkEVM 1. Deposit jsx const depositTx = await erc20Token.depositCustomERC20("1000000000000000000", "recipent address",true); const txHash = await depositTx.getTransactionHash(); console.log("Transaction Hash", txHash); 2. Claim deposit jsx const claimTx = await erc20.customERC20DepositClaim(""); const txHash = await claimTx.getTransactionHash(); console.log("claimed txHash", ctxHash); Bridge from zkEVM â†' Ethereum 1. Withdraw jsx const depositTx = await erc20Token.withdrawCustomERC20("1000000000000000000", "recipent address",true); const txHash = await depositTx.getTransactionHash(); console.log("Transaction Hash", txHash); 2. Claim withdrawal jsx const claimTx = await erc20.customERC20WithdrawExit(""); const txHash = await claimTx.getTransactionHash(); console.log("claimed txHash", ctxHash); Basic functions for error passing Below we provide the two basic functions used for error passing in each of the two directions: L1 --> L2 and L2 --> L1. Root to child (L1 â†' L2) jsx const bridgeTx = zkEvmClient.rootChainBridge.bridgeMessage(destinationNetwork: number, destinationAddress: string, forceUpdateGlobalExitRoot: boolean, permitData = '0x', option?: ITransactionOption); const claimTx = zkEvmClient.childChainBridge.claimMessage(smtProof: string[], smtProofRollup: string[], globalIndex: string, mainnetExitRoot: string, rollupExitRoot: string, originNetwork: number, originTokenAddress: string, destinationNetwork: number, destinationAddress: string, amount: TYPEAMOUNT, metadata: string, option: ITransactionOption); // proof can be fetched from the proof gen API Child to root (L2 â†' L1) jsx const bridgeTx = zkEvmClient.childChainBridge.bridgeMessage(destinationNetwork: number, destinationAddress: string, forceUpdateGlobalExitRoot: boolean, permitData = '0x', option?: ITransactionOption); const claimTx = zkEvmClient.rootChainBridge.claimMessage(smtProof: string[], smtProofRollup: string[], globalIndex: string, mainnetExitRoot: string, rollupExitRoot: string, originNetwork: number, originTokenAddress: string, destinationNetwork: number, destinationAddress: string, amount: TYPEAMOUNT, metadata: string, option: ITransactionOption); // proof can be fetched from the proof gen API # api3.md: !!! info "Content disclaimer" Please view the third-party content disclaimer here. Overview API3 is a collaborative project to deliver traditional API services to smart contract platforms in a decentralized and trust-minimized way. It is governed by a decentralized autonomous organization (DAO), namely the API3 DAO. !!! info "The API3 DAO" Read more about how The API3 DAO works. Click here Airnode Developers can use Airnode to request off-chain data inside their Smart Contracts on the Polygon PoS and Polygon zkEVM. An Airnode is a first-party oracle that pushes off-chain API data to your on-chain contract. Airnode lets API providers easily run their own first-party oracle nodes. That way, they can provide data to any on-chain dApp that's interested in their services, all without an intermediary. An on-chain smart contract makes a request in the RRP (Request Response Protocol) contract (AirnodeRrpV0.sol) that adds the request to the event logs. The Airnode then accesses the event logs, fetches the API data and performs a callback to the requester with the requested data. !airnode1 Requesting off-chain data by calling an Airnode Requesting off-chain data essentially involves triggering an Airnode and getting its response through your smart contract. The smart contract in this case would be the requester contract which will make a request to the desired off-chain Airnode and then capture its response. The requester calling an Airnode primarily focuses on two tasks: - Make the request - Accept and decode the response !airnode2{width=70%}

Here is an example of a basic requester contract to request data from an Airnode: solidity // SPDX-License-Identifier: MIT pragma solidity 0.8.9; import "@api3/airnode-protocol/contracts/rrp/requesters/RrpRequesterV0.sol"; import "@openzeppelin/contracts@4.9.5/access/Ownable.sol"; // A Requester that will return the requested data by calling the specified Airnode. contract Requester is RrpRequesterV0, Ownable { mapping(bytes32 => bool) public incomingFulfillments; mapping(bytes32 => int256) public fulfilledData; // Make sure you specify the right rrpAddress for your chain while deploying the contract. constructor(address rrpAddress) RrpRequesterV0(rrpAddress) {} // To receive funds from the sponsor wallet and send them to the owner. receive() external payable { payable(owner()).transfer(address(this).balance); } // The main makeRequest function that will trigger the Airnode request. function makeRequest(address airnode, bytes32 endpointId, address sponsor, address sponsorWallet, bytes calldata parameters) external { bytes32 requestId = airnodeRrp.makeFullRequest(airnode, // airnode address endpointId, // endpointId sponsor, // sponsor's address sponsorWallet, // sponsorWallet address(this), // fulfillAddress this.fulfill.selector, // fulfillFunctionId parameters // encoded API parameters); incomingFulfillments[requestId] = true; } function fulfill(bytes32 requestId, bytes calldata data) external onlyAirnodeRrp { require(incomingFulfillments[requestId], "No such request made"); delete incomingFulfillments[requestId]; int256 decodedData = abi.decode(data, (int256)); fulfilledData[requestId] = decodedData; } // To withdraw funds from the sponsor wallet to the contract. function withdraw(address airnode, address sponsorWallet) external onlyOwner { airnodeRrp.requestWithdrawal(airnode, sponsorWallet); } } The rrpAddress is the main airnodeRrpAddress. The RRP Contracts have already been deployed on-chain. You can check the address for all supported chains here. You can also try deploying it using Remix | Contract | Addresses | | :----------: | :--: | | AirnodeRrpV0 | 0xa0AD79D995DdeeB18a14eAef56A549A04e3Aa1Bd | Request parameters The makeRequest() function expects the following parameters to make a valid request. - airnode: Specifies the Airnode Address. - endpointId: Specifies which endpoint to be used. - sponsor and sponsorWallet: Specifies which wallet will be used to fulfill the request. - parameters: Specifies the API and Reserved Parameters (see Airnode ABI specifications for how these are encoded). Parameters can be encoded off-chain using @airnode-abi library. Response parameters The callback to the Requester contains two parameters: - requestId: First acquired when making the request and passed here as a reference to identify the request for which the response is intended. - data: In case of a successful response, this is the requested data which has been encoded and contains a timestamp in addition to other response data. Decode it using the decode() function from the abi object. !!! info "Note" Sponsors should not fund a sponsorWallet with more then they can trust the Airnode with, as the Airnode controls the private key to the sponsorWallet. The deployer of such Airnode undertakes no custody obligations, and the risk of loss or misuse of any excess funds sent to the sponsorWallet remains with the sponsor. Try deploying it on Remix! Using dAPIs - API3 datafeeds dAPIs are continuously updated streams of off-chain data, such as the latest cryptocurrency, stock and commodity prices. They can power various decentralized applications such as DeFi lending, synthetic assets, stablecoins, derivatives, NFTs and more. The data feeds are continuously updated by first-party oracles using signed data. dApp owners can read the on-chain value of any dAPI in real-time. Due to being composed of first-party data feeds, dAPIs offer security, transparency, cost-efficiency and scalability in a turn-key package. Apart from relying on deviation threshold and heartbeat configuration updates, unlike traditional data feeds, OEV Network enables dApps using dAPIs to auction off the right to update the data feeds to searcher bots. Searcher bots can bid for price updates through the OEV Network to update the data feeds. All the OEV proceeds go back to the dApp. The API3 Market enables users to connect to a dAPI and access the associated data feed services. !dapi-main To learn more about how dAPIs work, click here Subscribing to dAPIs The API3 Market lets users access dAPIs on Polygon Mainnet, Polygon Testnet Polygon zkEVM and Polygon zkEVM Testnet. Exploring, selecting and configuring your dAPI The API3 Market provides a list of all the dAPIs available across multiple chains including testnets. You can filter the list by mainnet or testnet chains. After selecting the chain, you can now search for a specific dAPI by name. Once selected, you will land on the details page (eg ETH/USD on Polygon zkEVM Testnet) where you can find more information about the dAPI. The current supported configurations for dAPIs are: | Deviation | Heartbeat | | --------- | --------- | | 0.25% | 24 hours | | 0.5% | 24 hours | | 1% | 24 hours | | 5% | 24 hours | !dapi-1 Activating your dAPI !!! note "Note" If a dAPI is already activated, make sure to check the expiration date and update parameters. You can update the parameters and extend the subscription by purchasing a new configuration. After selecting the dAPI and the configuration, you will be presented with an option to purchase the dAPI and activate it. Make sure to check the time and amount of the subscription. If everything looks good, click on Purchase.
!dapi-2{width=65%}

You can then connect your wallet and confirm the transaction. Once it's confirmed, you will be able to see the updated configuration for the dAPI. Getting the proxy address Once you are done configuring and activating the dAPI, you can now integrate it. To do so, click on the Integrate button on the dAPI details page. !dapi-5 You can now see the deployed proxy contract address. You can now use this to read from the configured dAPI. Reading from a dAPI Here's an example of a basic contract that reads from a dAPI. solidity // SPDX-License-Identifier: MIT pragma solidity 0.8.17; import "@openzeppelin/contracts@4.9.5/access/Ownable.sol"; import "@api3/contracts/api3-server-v1/proxies/interfaces/IProxy.sol"; contract DataFeedReaderExample is Ownable { // The proxy contract address obtained from the API3 Market UI. address public proxyAddress; // Updating the proxy contract address is a security-critical // action. In this example, only the owner is allowed to do so. function setProxyAddress(address proxyAddress) public onlyOwner { proxyAddress = proxyAddress; } function readDataFeed() external view returns (int224 value, uint256 timestamp) { // Use the IProxy interface to read a dAPI via its // proxy contract . (value, timestamp) = IProxy(proxyAddress).read(); // If you have any assumptions about value and timestamp, // make sure to validate them after reading from the proxy. } } - setProxyAddress() is used to set the address of the dAPI Proxy Contract. - readDataFeed() is a view function that returns the latest price of the set dAPI. You can read more about dAPIs here. Try deploying it on Remix! Using API3 QRNG API3 QRNG is a public utility we provide with the courtesy of Australian National University (ANU), Quintessence Labs and Quantum Blockchains. It is powered by an Airnode hosted by the QRNG Providers, meaning that it is a first-party service. It is served as a public good and is free of charge (apart from the gas costs), and it provides â€˜trueâ€™ quantum randomness via an easy-to-use solution when requiring RNG on-chain. To request randomness on-chain, the requester submits a request for a random number to AirnodeRrpV0. The QRNG Airnode gathers the request from the AirnodeRrpV0 protocol contract, retrieves the random number off-chain, and sends it back to AirnodeRrpV0. Once received, it performs a callback to the requester with the random number. Click here to check out the AirnodeRrpV0 contract addresses and available QRNG Providers on Polygon and Polygon zkEVM. Here is an example of a basic QrngRequester that requests a random number: solidity //SPDX-License-Identifier: MIT pragma solidity 0.8.9; import "@api3/airnode-protocol/contracts/rrp/requesters/RrpRequesterV0.sol"; import "@openzeppelin/contracts@4.9.5/access/Ownable.sol"; /// @title Example contract that uses Airnode RRP to access QRNG services contract QrngExample is RrpRequesterV0, Ownable { event RequestedUint256(bytes32 indexed requestId); event ReceivedUint256(bytes32 indexed requestId, uint256 response); event RequestedUint256Array(bytes32 indexed requestId, uint256 size); event ReceivedUint256Array(bytes32 indexed requestId, uint256[] response); event WithdrawalRequested(address indexed airnode, address indexed sponsorWallet); address public airnode; /// The address of the QRNG Airnode bytes32 public endpointIdUint256; /// The endpoint ID for requesting a single random number bytes32 public endpointIdUint256Array; /// The endpoint ID for requesting an array of random numbers address public sponsorWallet; /// The wallet that will cover the gas costs of the request uint256 public qrngUint256; /// The random number returned by the QRNG Airnode uint256[] public qrngUint256Array; /// The array of random numbers returned by the QRNG Airnode mapping(bytes32 => bool) public expectingRequestWithIdToBeFulfilled; constructor(address airnodeRrp) RrpRequesterV0(airnodeRrp) {} /// @notice Sets the parameters for making requests function setRequestParameters(address airnode, bytes32 endpointIdUint256, bytes32 endpointIdUint256Array, address sponsorWallet) external { airnode = airnode; endpointIdUint256 = endpointIdUint256; endpointIdUint256Array = endpointIdUint256Array; sponsorWallet = sponsorWallet; } /// @notice To receive funds from the sponsor wallet and send them to the owner. receive() external payable { payable(owner()).transfer(msg.value); emit WithdrawalRequested(airnode, sponsorWallet); } /// @notice Requests a uint256 /// @dev This request will be fulfilled by the contract's sponsor wallet, /// which means spamming it may drain the sponsor wallet. function makeRequestUint256() external { bytes32 requestId = airnodeRrp.makeFullRequest(airnode, endpointIdUint256, address(this), sponsorWallet, address(this), this.fulfillUint256.selector, ""); expectingRequestWithIdToBeFulfilled[requestId] = true; emit RequestedUint256(requestId); } /// @notice Called by the Airnode through the AirnodeRrp contract to /// fulfill the request function fulfillUint256(bytes32 requestId, bytes calldata data) external onlyAirnodeRrp { require(expectingRequestWithIdToBeFulfilled[requestId], "Request ID not known"); expectingRequestWithIdToBeFulfilled[requestId] = false; uint256 qrngUint256 = abi.decode(data, (uint256)); qrngUint256 = qrngUint256; // Do what you want with qrngUint256 here... emit ReceivedUint256(requestId, qrngUint256); } /// @notice Requests a uint256[] /// @param size Size of the requested array function makeRequestUint256Array(uint256 size) external { bytes32 requestId =

airnodeRrp.makeFullRequest(airnode, endpointIdUint256Array, address(this), sponsorWallet, address(this), this.fulfillUint256Array.selector, // Using Airnode ABI to encode the parameters abi.encode(bytes32("1u"), bytes32("size"), size)); expectingRequestWithIdToBeFulfilled[requestId] = true; emit RequestedUint256Array(requestId, size); } /// @notice Called by the Airnode through the AirnodeRrp contract to /// fulfill the request function fulfillUint256Array(bytes32 requestId, bytes calldata data) external onlyAirnodeRrp { require(expectingRequestWithIdToBeFulfilled[requestId], "Request ID not known"); expectingRequestWithIdToBeFulfilled[requestId] = false; uint256[] memory qrngUint256Array = abi.decode(data, (uint256[])); // Do what you want with qrngUint256Array here... qrngUint256Array = qrngUint256Array; emit ReceivedUint256Array(requestId, qrngUint256Array); } /// @notice Getter functions to check the returned value. function getRandomNumber() public view returns (uint256) { return qrngUint256; } function getRandomNumberArray() public view returns(uint256[] memory) { return qrngUint256Array; } /// @notice To withdraw funds from the sponsor wallet to the contract. function withdraw() external onlyOwner { airnodeRrp.requestWithdrawal(airnode, sponsorWallet); } } - The setRequestParameters() takes in airnode , endpointIdUint256, sponsorWallet and sets these parameters. You can get the Airnode address and the endpoint ID here. - The makeRequestUint256() function calls the airnodeRrp.makeFullRequest() function of the AirnodeRrpV0.sol protocol contract which adds the request to its storage and returns a requestId. - The targeted off-chain Airnode gathers the request and performs a callback to the requester with the random number. Try deploying it on Remix! You can try QRNG on the Polygon and Polygon zkEVM for free. Check out the all the QRNG Providers here. Click here to read more about API3 QRNG Additional resources Here are some additional developer resources - API3 Docs - API3 Market on Polygon - API3 Market on Polygon zkEVM - Get started with dAPIs - get started with QRNG - Github - Medium - YouTube # bandchain.md: !!! info "Content disclaimer" Please view the third-party content disclaimer here. Band Protocol allows you to query data from traditional web APIs and use it in the blockchain. Developers can make queries through BandChain, a cosmos-based blockchain for facilitating oracle requests and payment, and then use the data on the dApp through inter-chain communication. Integrating oracle data can be done in 3 simple steps: 1. Choosing the oracle scripts Oracle script is a hash that uniquely identifies the type of data to be requested from band-chain. These scripts are used as one of the parameters while making the oracle request. 2. Requesting data from BandChain This can be done in two ways: - Using the BandChain explorer You can click on the oracle script of your choice, and then from the Execute tab you can pass in the parameters and get the response from BandChain. The response will contain the result and also an EVM proof. This proof has to be copied and will be used in the final step. The BandChain docs for querying oracle using explorer are available here. !img Given above is an example of making an oracle request to get the random number values. The value 100 is passed to the maxrange parameter of the oracle request. We get a hash in response. Clicking on this hash will show us the complete details of the response. - Using the BandChain-Devnet JS library You can query BandChain directly using the BandChain-Devnet library. When queried, it gives an EVM proof in the response. This proof can be used for the final step of BandChain integration. The BandChain docs for querying oracle using BandChain-Devnet JS Library is available here. The request payload for the random number oracle will look like this. Make sure the request body is passed in application/json format. 3. Using the data in smart contracts The final step is to deploy a validation contract and store the responses from the oracle request into the validation contracts state variables. Once these state variables are set, they can be accessed as and when required by the dApp. Also these state variables can be updated with new values by querying the oracle scripts again from the dApp. Given below is a validation contract that stores the random number value using the random number oracle script. jsx pragma solidity 0.5.14; pragma experimental ABIEncoderV2; import "BandChainLib.sol"; import "IBridge.sol"; contract SimplePriceDatabase { using BandChainLib for bytes; bytes32 public codeHash; bytes public params; IBridge public bridge; uint256 public latestPrice; uint256 public lastUpdate; constructor(bytes32 codeHash , bytes memory params, IBridge bridge) public { codeHash = codeHash; params = params; bridge = bridge; } function update(bytes memory reportPrice) public { IBridge.VerifyOracleDataResult memory result = bridge.relayAndVerify(reportPrice); uint64[] memory decodedInfo = result.data.toUint64List(); require(result.codeHash == codeHash, "INVALIDCODEHASH"); require(keccak256(result.params) == keccak256(params), "INVALIDPARAMS"); require(uint256(decodedInfo[1]) > lastUpdate, "TIMESTAMPMUSTBEOLDERTHANTHELASTUPDATE"); latestPrice = uint256(decodedInfo[0]); lastUpdate = uint256(decodedInfo[1]); } } When deploying, 3 parameters have to be passed. The first parameter is the codeHash which is the oracle script hash. The second parameter is the oracle script request parameters object. This has to be passed in bytes format. BandChain provides a REST API for converting the parameter JSON object to bytes format. The API details can be found here. A 0x has to be appended to the response received from this API. The third parameter is the contract address of the BandChain contract that is already deployed on the Polygon network. Band Protocol supports Polygon TestnetV3: 0x3ba819b03fb8d34995f68304946eefa6dcff7cbf. Another thing to note is that the validation contract should import the helper library and interface which is called BandChainLib.sol and IBridge.sol respectively. Once the validation contract is deployed, the state variables can be accessed by querying from a dApp. Similarly multiple validation contracts can be created for different in-built oracle scripts. The IBridge interface has a method called relayAndVerify() that verifies the values being updated each time in the validation contract. The update() method in the validation contract has the logic to update the state variables. The EVM proof obtained from querying the oracle script has to be passed to the update() method. Each time a value is updated, the BandChain contract deployed on Polygon verifies the data before storing it in the contract state variable. The BandChain provides a decentralized network of oracles that can be used by dApps to boost their smart contract logic. The BandChain docs on deploying the contract, storing the values, and updating them can be found here. # bandstandarddataset.md: !!! info "Content disclaimer" Please view the third-party content disclaimer here. Developers building on Polygon can now leverage Band Protocol's decentralized oracle infrastructure. With Band Protocol's oracle, they now have access to various cryptocurrency price data to integrate into their applications. Supported tokens Currently, the list of supported symbols can be found at https://data.bandprotocol.com/. Going forward, this list will continue to expand based on developer needs and community feedback. Price pairs The following methods can work with any combination of base/quote token pair, as long as the base and quote symbols are supported by the dataset. Querying prices Currently, there are two methods for developers to query prices from Band Protocol's oracle: through Band's StdReference smart contract on Polygon and through their bandchain.js JavaScript helper library. Solidity smart contract To query prices from Band Protocol's oracle, a smart contract should reference Band's StdReference contract, specifically the getReferenceData and getReferenceDatabulk methods. getReferenceData takes two strings as the inputs, the base and quote symbol, respectively. It then queries the StdReference contract for the latest rates for those two tokens, and returns a ReferenceData struct, shown below. c++ struct ReferenceData { uint256 rate; // base/quote exchange rate, multiplied by 1e18. uint256 lastUpdatedBase; // UNIX epoch of the last time when base price gets updated. uint256 lastUpdatedQuote; // UNIX epoch of the last time when quote price gets updated. } getReferenceDataBulk instead takes two lists, one of the base tokens, and one of the quotes. It then proceeds to similarly query the price for each base/quote pair at each index, and returns an array of ReferenceData structs. For example, if we call getReferenceDataBulk with ['BTC','BTC','ETH'] and ['USD','ETH','BNB'], the returned ReferenceData array will contain information regarding the pairs: - BTC/USD - BTC/ETH - ETH/BNB Contract addresses | Blockchain | Contract Address | | -------------------- | :--: | | Polygon (Test) | 0x56e2898e0ceff0d1222827759b56b28ad812f92f | BandChain.JS Band's node helper library bandchain.js also supports a similar getReferenceData function. This function takes one argument, a list of token pairs to query the result. It then returns a list of corresponding rate values. Example usage The code below shows an example usage of the function: javascript const { Client } = require('@bandprotocol/bandchain.js'); // BandChain's REST Endpoint const endpoint = 'https://rpc.bandchain.org'; const client = new Client(endpoint); // This example demonstrates how to query price data from // Band's standard dataset async function exampleGetReferenceData() { const rate = await client.getReferenceData(['BTC/ETH','BAND/EUR']); return rate; } (async () => { console.log(await exampleGetReferenceData()); })(); The corresponding result will then be similar to: bash $ node index.js [{ pair: 'BTC/ETH', rate: 30.998744363906173, updatedAt: { base: 1615866954, quote: 1615866954 }, requestID: { base: 2206590, quote: 2206590 } }, { pair: 'BAND/EUR', rate: 10.566138918332376, updatedAt: { base: 1615866845, quote: 1615866911 }, requestID: { base: 2206539, quote: 2206572 } }] For each pair, the following information will be returned: - pair: The base/quote symbol pair string. - rate: The resulting rate of the given pair. - updated: The timestamp at which the base and quote symbols was last updated on BandChain. For USD, this will be the current timestamp. - rawRate: This object consists of two parts. - value is the BigInt value of the actual rate, multiplied by 10^decimals. - decimals is then the exponent by which rate was multiplied by to get rawRate. Example usage This contract demonstrates an example of using Band's StdRefence contract and the getReferenceData function. # chainlink.md: !!! info "Content disclaimer" Please view the third-party content disclaimer here. Chainlink enables your contracts to access any external data source, through a decentralized oracle network. Whether your contract requires sports results, the latest weather, or any other publicly available data, Chainlink provides the tools required for your contract to consume it. Decentralized data One of Chainlink's most powerful features is already decentralized, aggregated, and ready to be digested on-chain data on most of the popular cryptocurrencies. These are known as Chainlink Data Feeds. Here is a working example of a contract that pulls the latest price of MATIC in USD on the Mumbai Testnet. All you need to do is swap out the address with any address of a data feed that you wish, and you can start digesting price information. solidity pragma solidity ^0.6.7; import "@chainlink/contracts/src/v0.6/interfaces/AggregatorV3Interface.sol"; contract PriceConsumerV3 { AggregatorV3Interface internal priceFeed; / Network: Mumbai Testnet Aggregator: MATIC/USD Address: 0xd0D5e3DB44DE05E9F294BB0a3bEEaF030DE24Ada / constructor() public { priceFeed = AggregatorV3Interface(0xd0D5e3DB44DE05E9F294BB0a3bEEaF030DE24Ada); } / Returns the latest price / function getLatestPrice() public view returns (int) { (uint80 roundID, int price, uint startedAt, uint timeStamp, uint80 answeredInRound) = priceFeed.latestRoundData(); return price; } } Request and receive cycle Chainlink's Request and Receive cycle enables your smart contracts to make a request to any external API and consume the response. To implement it, your contract needs to define two functions: 1. One to request the data, and 2. Another to receive the response. To request data, your contract builds a request object which it provides to an oracle. Once the oracle has reached out to the API and parsed the response, it will attempt to send the data back to your contract using the callback function defined in your smart contract. Uses 1. Chainlink Data Feeds These are decentralized data reference points already aggregated on-chain, and the quickest, easiest, and cheapest way to get data from the real world. Currently supports some of the most popular cryptocurrency and fiat pairs. For working with Data Feeds, use the Polygon Data Feeds from the Chainlink documentation. 2. Chainlink Verifiable Randomness Function Get provably random numbers, where the random number is cryptographically guaranteed to be random. For working with Chainlink VRF, use the Polygon VRF addresses from the Chainlink documentation. 3. Chainlink API Calls How to configure your smart contract to work with traditional APIs, and customize to get any data, send any requests over the internet, and more. Code example To interact with external APIs, your smart contract should inherit from ChainlinkClient.sol, which is a contract designed to make processing requests easy. It exposes a struct called Chainlink.Request, which your contract should use to build the API request. The request should define the oracle address, job id, fee, adapter parameters, and the callback function signature. In this example, the request is built in the requestEthereumPrice function. fulfill is defined as the callback function. solidity pragma solidity ^0.6.0; import "@chainlink/contracts/src/v0.6/ChainlinkClient.sol"; contract APIConsumer is ChainlinkClient { uint256 public price; address private oracle; bytes32 private jobId; uint256 private fee; / Network: Polygon Mumbai Testnet Oracle: 0x58bbdbfb6fca3129b91f0dbe372098123b38b5e9 Job ID: da20aae0e4c843f6949e5cb3f7cfe8c4 LINK address: 0x326C977E6efc84E512bB9C30f76E30c160eD06FB Fee: 0.01 LINK / constructor() public { setChainlinkToken(0x326C977E6efc84E512bB9C30f76E30c160eD06FB); oracle = 0x58bbdbfb6fca3129b91f0dbe372098123b38b5e9; jobId = "da20aae0e4c843f6949e5cb3f7cfe8c4"; fee = 10 16; // 0.01 LINK } / Create a Chainlink request to retrieve API response, find the target price data, then multiply by 100 (to remove decimal places from price). / function requestBTCCNYPrice() public returns (bytes32 requestId) { Chainlink.Request memory request = buildChainlinkRequest(jobId, address(this), this.fulfill.selector); // Set the URL to perform the GET request on // NOTE: If this oracle gets more than 5 requests from this job at a time, it will not return. request.add("get", "https://www.alphavantage.co/query?function=CURRENCYEXCHANGERATE&fromcurrency=BTC&tocurrency=CNY&apikey=demo"); // Set the path to find the desired data in the API response, where the response format is: // { // "Realtime Currency Exchange Rate": { // "1. FromCurrency Code": "BTC", // "2. FromCurrency Name": "Bitcoin", // "3. ToCurrency Code": "CNY", // "4. ToCurrency Name": "Chinese Yuan", // "5. Exchange Rate": "207838.88814500", // "6. Last Refreshed": "2021-01-26 11:11:07", // "7. Time Zone": "UTC", // "8. Bid Price": "207838.82343000", // "9. Ask Price": "207838.88814500" // } // } string[] memory path = new string[](2); path[0] = "Realtime Currency Exchange Rate"; path[1] = "5. Exchange Rate"; request.addStringArray("path", path); // Multiply the result by 10000000000 to remove decimals request.addInt("times", 10000000000); // Sends the request return sendChainlinkRequestTo(oracle, request, fee); } / Receive the response in the form of uint256 / function fulfill(bytes32 requestId, uint256 price) public recordChainlinkFulfillment(requestId) { price = price; } } Mainnet Polygon LINK token To get mainnet Polygon LINK token from the Ethereum Mainnet, you must follow a 2-step process. 1. Bridge your LINK using the PoS bridge. 2. Swap the LINK for the ERC677 version via the Pegswap, deployed by the Chainlink. The Polygon bridge brings over an ERC20 version of LINK, and LINK is an ERC677, so we just have to update it with this swap. Addresses There are currently only a few operational Chainlink oracles on the Polygon Mumbai Testnet. You can always run one yourself too, and list it on the Chainlink Marketplace. Oracle: 0xb33D8A4e62236eA91F3a8fD7ab15A95B9B7eEc7D LINK: 0x326C977E6efc84E512bB9C30f76E30c160eD06FB To obtain LINK on Mumbai Testnet, head to the Polygon faucet here. Supported APIs Chainlink's Request and Receive cycle is flexible enough to call any public API, so long as the request parameters are correct and the response format is known. For example, if the response object from a URL we want to fetch from is formatted like this: {"USD":243.33}, the path is simple: "USD". If an API responds with a complex JSON object, the path parameter would need to specify where to retrieve the desired data, using a dot delimited string for nested objects. For example, consider the following response: json { "Prices":{ "USD":243.33 } } This would require the following path: "Prices.USD". If there are spaces in the strings, or the strings are quite long, we can use the syntax shown in the example above, where we pass them all as a string array. json string[] memory path = new string[](2); path[0] = "Prices"; path[1] = "USD"; request.addStringArray("path", path); What are job IDs for? You may have noticed that our example uses a jobId parameter when building the request. Jobs are comprised of a sequence of instructions that an oracle is configured to run. In the code example above, the contract makes a request to the oracle with the job ID: da20aae0e4c843f6949e5cb3f7cfe8c4. This particular job is configured to do the following: Make a GET request Parse the JSON response Multiply the value by x Convert the value to uint Submit to the chain This is why our contract adds in the URL, the path of where to find the desired data in the JSON response, and the times amount to the request; using the request.add statements. These instructions are facilitated by what's known as Adapters, in the oracle. Every request to an oracle must include a specific job ID. Here is the list of jobs that the Polygon oracle is configured to run. | Name | Return Type | ID | Adapters | |-----|--------|------|-------| | HTTP GET | uint256 | da20aae0e4c843f6949e5cb3f7cfe8c4 | httpget
jsonparse
multiply
ethuint256
ethtx | | HTTP GET | int256 | e0c76e45462f4e429ba32c114bfbf5ac | httpget
jsonparse
multiply
ethint256
ethtx | | HTTP GET | bool | 999539ec63414233bdc989d8a8ff10aa | httpget
jsonparse
ethbool
ethtx | | HTTP GET | bytes32 | a82495a8fd5b4cb492b17dc0cc31a4fe | httpget
jsonparse
ethbytes32
ethtx | | HTTP GET | string | 7d80a6386ef543a3abb52817f6707e3b | httpget
jsonparse
ethstring
ethtx | | HTTP POST | bytes32 | a82495a8fd5b4cb492b17dc0cc31a4fe | httppost
jsonparse
ethbytes32
ethtx | The complete Chainlink API reference can be found here. # chronicle.md: !!! info "Content disclaimer" Please view the third-party content disclaimer here. Chronicle Chronicle Protocol is a novel Oracle solution that has exclusively secured over $10B in assets for MakerDAO and its ecosystem since 2017. Chronicle overcomes the current limitations of transferring data on-chain by developing scalable, cost-efficient, decentralized, and verifiable Oracles, rewriting the rulebook on data transparency and accessibility. Querying the price of MATIC using Chronicle Chronicle contracts are read-protected by a whitelist, meaning you won't be able to read them on-chain without your address being added to the whitelist. On the Testnet, users can add themselves to the whitelist through the SelfKisser contract, a process playfully referred to as "kissing" themselves. For access to production Oracles on the Mainnet, please open a support ticket on Discord in the ðŸ†˜support channel. For the deployment addresses, please check out the Dashboard. solidity // SPDX-License-Identifier: MIT pragma solidity ^0.8.16; / @title OracleReader @notice A simple contract to read from Chronicle oracles @dev To see the full repository, visit https://github.com/chronicleprotocol/OracleReader-Example. @dev Addresses in this contract are hardcoded for the zkEVM Testnet. For other supported networks, check the https://chroniclelabs.org/dashboard/oracles. / contract OracleReader { / @notice The Chronicle oracle to read from. ChronicleMATICUSD1 - 0x55a07a60cd9ed198B5Ba4360FF9800eBb6667388 Network: zkEVM Testnet / IChronicle public chronicle = IChronicle(address(0x55a07a60cd9ed198B5Ba4360FF9800eBb6667388)); / @notice The SelfKisser granting access to Chronicle oracles. SelfKisser1:0x0Dcc19657007713483A5cA76e6A7bbe5f56EA37d Network: zkEVM Testnet / ISelfKisser public selfKisser =

ISelfKisser(address(0x0Dcc19657007713483A5cA76e6A7bbe5f56EA37d)); constructor() { // Note to add address(this) to chronicle oracle's whitelist. // This allows the contract to read from the chronicle oracle. selfKisser.selfKiss(address(chronicle)); } / @notice Function to read the latest data from the Chronicle oracle. @return val The current value returned by the oracle. @return age The timestamp of the last update from the oracle. / function read() external view returns (uint256 val, uint256 age) { (val, age) = chronicle.readWithAge(); } } // Copied from chronicle-std. interface IChronicle { / @notice Returns the oracle's current value. @dev Reverts if no value set. @return value The oracle's current value. / function read() external view returns (uint256 value); / @notice Returns the oracle's current value and its age. @dev Reverts if no value set. @return value The oracle's current value using 18 decimals places. @return age The value's age as a Unix Timestamp . / function readWithAge() external view returns (uint256 value, uint256 age); } // Copied from self-kisser. interface ISelfKisser { /// @notice Kisses caller on oracle oracle. function selfKiss(address oracle) external; } More examples For more examples on integrating Chronicle Oracles, please check the documentation portal. # diadataoracles.md: !!! info "Content disclaimer" Please view the third-party content disclaimer here. dApp developers who want to leverage DIA oracles can access the published data on Polygon. DIA offers cryptocurrency and data about traditional financial assets. For more details click here. # getting-started.md: There is often a need to access information from the outside world that is relevant to the contractual agreement. But smart contracts can't access data from outside of their blockchain network. An oracle is a way for a blockchain or smart contract to interact with external data. With blockchains being deterministic one-way streets, an oracle is the path between off-chain and on-chain events. These oracles are services that send and verify real world occurrences and submit this information to smart contracts, triggering state changes on the blockchain. Inbound oracles bring off-chain or real-world data to the blockchain, whereas their outbound cousins do the opposite: they inform an entity outside the blockchain of an event that occurred on it. Resources 1. What is the blockchain oracle problem? 1. What is a blockchain oracle. 1. Tutorial: Getting decentralized price data. # optimisticoracle.md: !!! info "Content disclaimer" Please view the third-party content disclaimer here. UMA's Optimistic Oracle allows contracts to quickly request and receive any kind of data. UMA's oracle system is comprised of two core components: 1. Optimistic oracle 2. Data verification mechanism (DVM) Optimistic oracle UMA's Optimistic Oracle allows contracts to quickly request and receive price information. The Optimistic Oracle acts as a generalized escalation game between contracts that initiate a price request and UMA's dispute resolution system known as the Data Verification Mechanism (DVM). Prices proposed by the Optimistic Oracle will not be sent to the DVM unless it is disputed. This enables contracts to obtain price information within any pre-defined length of time without writing the price of an asset on-chain. Data verification mechanism (DVM) If a dispute is raised, a request is sent to the DVM. All contracts built on UMA use the DVM as a backstop to resolve disputes. Disputes sent to the DVM will be resolved 48 hours after UMA tokenholders vote on the price of the asset at a given time. Contracts on UMA do not need to use the Optimistic Oracle unless it requires a price of an asset faster than 48 hours. The Data Verification Mechanism (DVM) is the dispute resolution service for contracts built on UMA Protocol. The DVM is powerful because it encompasses an element of human judgment to ensure contracts are securely and correctly managed when issues arise from volatile (and sometimes manipulatable) markets. Optimistic oracle interface The Optimistic Oracle is used by financial contracts or any third party to retrieve prices. Once a price is requested, anyone can propose a price in response. Once proposed, the price goes through a liveness period where anyone can dispute the proposed price and send the disputed price to the UMA DVM for settlement. !!! info This section explains how different participants can interact with the Optimistic Oracle. To view the most updated mainnet, kovan or L2 deployments of the Optimistic Oracle contracts, refer to the production addresses. There are twelve methods that make up the Optimistic Oracle interface. - requestPrice - proposePrice - disputePrice - settle - hasPrice - getRequest - settleAndGetPrice - setBond - setCustomLiveness - setRefundOnDispute - proposePriceFor - disputePriceFor requestPrice Requests a new price. This must be for a registered price identifier. Note that this is called automatically by most financial contracts that are registered in the UMA system, but can be called by anyone for any registered price identifier. For example, the Expiring Multiparty (EMP) contract calls this method when its expire method is called. Parameters: - identifier: price identifier being requested. - timestamp: timestamp of the price being requested. - ancillaryData: ancillary data representing additional args being passed with the price request. - currency: ERC20 token used for payment of rewards and fees. Must be approved for use with the DVM. - reward: reward offered to a successful proposer. Will be paid by the caller. Note: this can be 0. proposePrice Proposes a price value for an existing price request. Parameters: - requester: sender of the initial price request. - identifier: price identifier to identify the existing request. - timestamp: timestamp to identify the existing request. - ancillaryData: ancillary data of the price being requested. - proposedPrice: price being proposed. disputePrice Disputes a price value for an existing price request with an active proposal. Parameters: - requester: sender of the initial price request. - identifier: price identifier to identify the existing request. - timestamp: timestamp to identify the existing request. - ancillaryData: ancillary data of the price being requested. settle Attempts to settle an outstanding price request. Will revert if it canâ€™t be settled. Parameters: - requester: sender of the initial price request. - identifier: price identifier to identify the existing request. - timestamp: timestamp to identify the existing request. - ancillaryData: ancillary data of the price being requested. hasPrice Checks if a given request has resolved or been settled (i.e the optimistic oracle has a price). Parameters: - requester: sender of the initial price request. - identifier: price identifier to identify the existing request. - timestamp: timestamp to identify the existing request. - ancillaryData: ancillary data of the price being requested. getRequest Gets the current data structure containing all information about a price request. Parameters: - requester: sender of the initial price request. - identifier: price identifier to identify the existing request. - timestamp: timestamp to identify the existing request. - ancillaryData: ancillary data of the price being requested. settleAndGetPrice Retrieves a price that was previously requested by a caller. Reverts if the request is not settled or settleable. Note: this method is not view so that this call may actually settle the price request if it hasnâ€™t been settled. Parameters: - identifier: price identifier to identify the existing request. - timestamp: timestamp to identify the existing request. - ancillaryData: ancillary data of the price being requested. setBond Set the proposal bond associated with a price request. Parameters: - identifier: price identifier to identify the existing request. - timestamp: timestamp to identify the existing request. - ancillaryData: ancillary data of the price being requested. - bond: custom bond amount to set. setCustomLiveness Sets a custom liveness value for the request. Liveness is the amount of time a proposal must wait before being auto-resolved. Parameters: - identifier: price identifier to identify the existing request. - timestamp: timestamp to identify the existing request. - ancillaryData: ancillary data of the price being requested. - customLiveness: new custom liveness. setRefundOnDispute Sets the request to refund the reward if the proposal is disputed. This can help to "hedge" the caller in the event of a dispute-caused delay. Note: in the event of a dispute, the winner still receives the otherâ€™s bond, so there is still profit to be made even if the reward is refunded. Parameters: - identifier: price identifier to identify the existing request. - timestamp: timestamp to identify the existing request. - ancillaryData: ancillary data of the price being requested. disputePriceFor Disputes a price request with an active proposal on another address' behalf. Note: this address will receive any rewards that come from this dispute. However, any bonds are pulled from the caller. Parameters: - disputer: address to set as the disputer. - requester: sender of the initial price request. - identifier: price identifier to identify the existing request. - timestamp: timestamp to identify the existing request. - ancillaryData: ancillary data of the price being requested. proposePriceFor Proposes a price value on another address' behalf. Note: this address will receive any rewards that come from this proposal. However, any bonds are pulled from the caller. Parameters: - proposer: address to set as the proposer. - requester: sender of the initial price request. - identifier: price identifier to identify the existing request. - timestamp: timestamp to identify the existing request. - ancillaryData: ancillary data of the price being requested. - proposedPrice: price being proposed. Integrating the optimistic oracle This demo will set up an OptimisticDepositBox contract which custodies a userâ€™s ERC-20 token balance. On a local testnet blockchain, the user will deposit wETH (Wrapped Ether) into the contract and withdraw wETH denominated in USD. For example, if the user wants to withdraw $10,000 USD of wETH, and the ETH/USD exchange rate is $2,000, they would withdraw 5 wETH. - The user links the OptimisticDepositBox with one of the price identifiers enabled on the DVM. - The user deposits wETH into the OptimisticDepositBox and register it with the ETH/USD price identifier. - The user can now withdraw a USD-denominated amount of wETH from their DepositBox via smart contract calls, with the Optimistic Oracle enabling optimistic on-chain pricing. In this example, the user would not have been able to transfer USD-denominated amounts of wETH without referencing an off-chain ETH/USD price feed. The Optimistic Oracle therefore enables the user to "pull" a reference price. Unlike price requests to the DVM, a price request to the Optimistic Oracle can be resolved within a specified liveness window if there are no disputes, which can be significantly shorter than the DVM voting period. The liveness window is configurable, but is typically two hours, compared to 2-3 days for settlement via the DVM. The price requester is not currently required to pay fees to the DVM. The requester can offer a reward for the proposer who responds to a price request, but the reward value is set to 0 in this example. The price proposer posts a bond along with their price, which will be refunded if the price is not disputed, or if a dispute is resolved in the proposer's favor. Otherwise, this bond is used to pay the final fee to the DVM and pay a reward to a successful disputer. In the demo, the requester does not require an additional bond from the price proposer, so the total bond posted is equal to the wETH final fee currently 0.2 wETH. See the proposePriceFor function in the OptimisticOracle contract for implementation details. Running the demo 1. Ensure that you have followed all the prerequisite setup steps here. 2. Run a local Ganache instance (i.e. not Kovan/Ropsten/Rinkeby/Mainnet) with yarn ganache-cli --port 9545 3. In another window, migrate the contracts by running the following command: bash yarn truffle migrate --reset --network test 1. To deploy the OptimisticDepositBox contract and go through a simple user flow, run the following demo script from the root of the repo: bash yarn truffle exec ./packages/core/scripts/demo/OptimisticDepositBox.js --network test You should see the following output: 1. Deploying new OptimisticDepositBox - Using wETH as collateral token - Pricefeed identifier for ETH/USD is whitelisted - Collateral address for wETH is whitelisted - Deployed an OptimisticOracle - Deployed a new OptimisticDepositBox 2. Minting ERC20 to user and giving OptimisticDepositBox allowance to transfer collateral - Converted 10 ETH into wETH - User's wETH balance: 10 - Increased OptimisticDepositBox allowance to spend wETH - Contract's wETH allowance: 10 3. Depositing ERC20 into the OptimisticDepositBox - Deposited 10 wETH into the OptimisticDepositBox - User's deposit balance: 10 - Total deposit balance: 10 - User's wETH balance: 0 4. Withdrawing ERC20 from OptimisticDepositBox - Submitted a withdrawal request for 10000 USD of wETH - Proposed a price of 2000000000000000000000 ETH/USD - Fast-forwarded the Optimistic Oracle and Optimistic Deposit Box to after the liveness window so we can settle. - New OO time is [fast-forwarded timestamp] - New ODB time is [fast-forwarded timestamp] - Executed withdrawal. This also settles and gets the resolved price within the withdrawal function. - User's deposit balance: 5 - Total deposit balance: 5 - User's wETH balance: 5 Explaining the contract functions The OptimisticDepositBox contract code shows how to interact with the Oracle. The constructor function includes a finderAddress argument for the UMA Finder contract, which maintains a registry of the OptimisticOracle address, approved collateral and price identifier whitelists, and other important contract addresses. This allows the constructor to check that the collateral type and price identifier are valid, and allows the OptimisticDepositBox to find and interact with the OptimisticOracle later. The requestWithdrawal function includes an internal call to the OptimisticOracle requesting the ETH/USD price. Once it's returned, the user can call executeWithdrawal to complete the withdrawal. There is much more information and explanation in the code comments, so please take a look if you're interested in learning more. Additional resources Here are some additional resources regarding the UMA DVM: - Technical architecture. - Economic architecture. - Blog post on UMAâ€™s DVM design. - Whitepaper on UMAâ€™s DVM design. - Research repo for optimal fee policy. - UMIP repo for governance proposals. # ora.md: !!! info "Content disclaimer" Please view the third-party content disclaimer here. ORA is Ethereum's Trustless AI. ORA is the verifiable oracle protocol that brings AI and complex compute onchain. ORAâ€™s main product, Onchain AI Oracle (OAO), brings AI onchain. ORA breaks down the limitations of smart contracts by offering verifiable AI inference, so developers can innovate freely. OAO Quickstart This quickstart is designed to help you to build a smart-contract on Polygon able to interact with OAO. You can find more details in our docs quickstart. Workflow 1. The user contract sends the AI request to OAO on Polygon, by calling requestCallback function on the OAO contract. 2. Each AI request will initiate an opML inference. 3. OAO will emit a requestCallback event which will be collected by opML node. 4. opML node will run the AI inference, and then upload the result on Polygon, waiting for the challenge period. 1. During the challenge period, the opML validators will check the result, and challenge it if the submitted result is incorrect. 2. If the submitted result is successfully challenged by one of the validators, the submitted result will be updated on Polygon. 3. After the challenge period, the submitted result on chain is finalized. 5. When the result is uploaded or updated on Polygon, the provided AI inference in opML will be dispatched to the user's smart contract via its specific callback function. Integration Overview To integrate with OAO, you will need to write your own contract. To build with AI models of OAO, we provided an example of contract using OAO: Prompt. Smart Contract Integration 1. Inherit AIOracleCallbackReceiver in your contract and bind with a specific OAO address: solidity constructor(IAIOracle aiOracle) AIOracleCallbackReceiver(aiOracle) {} 2. Write your callback function to handle the AI result from OAO. Note that only OAO can call this function: solidity function aiOracleCallback(uint256 requestId, bytes calldata output, bytes calldata callbackData) external override onlyAIOracleCallback() 3. When you want to initiate an AI inference request, call OAO as follows: solidity aiOracle.requestCallback(modelId, input, address(this), gaslimit, callbackData); Reference 2 models are available on Polygon: Stable Diffusion (ID: 50), Llama3 8B Instruct (ID: 11). Prompt and SimplePrompt are both example smart contracts interacted with OAO. For simpler application scenarios (eg. Prompt Engineering based AI like GPTs), you can directly use Prompt or SimplePrompt. SimplePrompt saves gas by only emitting the event without storing historical data. Polygon PoS Mainnet: - OAO Proxy: 0x0A0f4321214BB6C7811dD8a71cF587bdaF03f0A0 - Prompt: 0xC20DeDbE8642b77EfDb4372915947c87b7a526bD - SimplePrompt: 0xC3287BDEF03b925A7C7f54791EDADCD88e632CcD Useful links: - Read ORA documentation - Join our Discord where our team can help you - Follow us on X # pyth.md: Overview The Pyth network is the largest first-party financial oracle network, delivering real-time market data to over 40 blockchains securely and transparently. The network comprises some of the worldâ€™s largest exchanges, market makers, and financial services providers; publishing proprietary price-data on-chain for aggregation and distribution to smart contract applications. Using Pyth network The Pyth network introduces an innovative low-latency pull oracle design, where users can pull price updates on-chain when needed, enabling everyone in the blockchain environment to access that data point. Developers on Polygon have permissionless access to any of Pythâ€™s 350+ price feeds for equities, ETFs, commodities, foreign exchange pairs, and cryptocurrencies. Here is a working example of a contract that fetches the latest price of MATIC/USD on the Polygon network. You have to pass Pyth's contract address for Polygon mainnet/testnet and the desired price feed id to fetch the latest price. solidity // SPDX-License-Identifier: UNLICENSED pragma solidity ^0.8.13; import "@pythnetwork/pyth-sdk-solidity/IPyth.sol"; contract MyFirstPythContract { IPyth pyth; constructor(address pyth) { pyth = IPyth(pyth); } function fetchPrice(bytes[] calldata pythPriceUpdate, bytes32 priceFeed) public payable returns (int64) { uint updateFee = pyth.getUpdateFee(pythPriceUpdate); pyth.updatePriceFeeds{value: updateFee}(pythPriceUpdate); // Fetch the latest price PythStructs.Price memory price = pyth.getPrice(priceFeed); return price.price; } } Here you can fetch the updateData from our Hermes feed, which listens to Pythnet and Wormhole for price updates; or you can use the pyth-evm-js SDK. This package provides utilities for consuming prices from the Pyth network oracle using Solidity. Also, it contains the Pyth Interface ABI that you can use in your libraries to communicate with the Pyth contract. We recommend following the consumer best practices when consuming Pyth data. For more information, check out the official Pyth documentation. There are details on the various functions available for interacting with the Pyth smart contract in the API Reference section. Pyth on Polygon The Pyth Network smart contract is available at the following address: - Mainnet: 0xff1a0f4744e8582DF1aE09D5611b887B6a12925C. - Amoy: 0x2880aB155794e7179c9eE2e38200202908C17B43 Additionally, click to access the Pyth price-feed IDs. Developers and community The Pyth network provides additional tools to developers, such as TradingView Integration, or the Gelato web3 functions. If you have any questions or issues, contact us on the following platforms: - Telegram - Discord - Website # razor.md: !!! info "Content disclaimer" Please view the third-party content disclaimer here. Razor network is a decentralized oracle providing data to the smart contracts using a network of stakers. The network consists of validators who lock in their tokens as a stake and provide data to the network. The honest validators are rewarded and those who report incoherently are penalized. The core of Razor Network is a set of smart contracts that can run on any Ethereum-compatible blockchain. Razor relies on the underlying blockchain for providing certain properties such as censorship resistance, security from network partition attacks, etc. Usage This oracle network is currently running on the Polygon mainnet and testnet. To get started with connecting to the data feed, check out this tutorial. The Razor Network can be explored using Razorscan. # supra.md: !!! info "Content disclaimer" Please view the third-party content disclaimer here. Supra is a novel, high-throughput oracle & intralayer: A vertically integrated toolkit of cross-chain solutions (data oracles, asset bridges, automation network, and more) that interlink all blockchains, public (L1s and L2s) or private (enterprises). Supra provides decentralized oracle price feeds that can be used for on-chain and off-chain use-cases such as spot and perpetual DEXes, lending protocols, and payments protocols. Supraâ€™s oracle chain and consensus algorithm makes it the fastest-to-finality oracle provider, with layer-1 security guarantees. The pull oracle has a sub-second response time. Aside from speed and security, Supraâ€™s rotating node architecture gathers data from 40+ data sources and applies a robust calculation methodology to get the most accurate value. The node provenance on the data dashboard also provides a fully transparent historical audit trail. Supraâ€™s Distributed Oracle Agreement (DORA) paper was accepted into ICDCS 2023, the oldest distributed systems conference. Check out our developer docs. # tellor.md: !!! info "Content disclaimer" Please view the third-party content disclaimer here. Tellor is an oracle that provides censorship resistant data that is secured by simple crypto-economic incentives. Data can be provided by anyone and checked by everyone. Tellorâ€™s flexible structure can provide any data at any time interval to allow for easy experimentation/innovation. (Soft) prerequisites We're assuming the following about your coding skill-level to focus on the oracle aspect. Assumptions: - You can navigate a terminal. - You have npm installed. - You know how to use npm to manage dependencies. Tellor is a live and open-sourced oracle ready for implementation. This beginner's guide is here to showcase the ease with which one can get up and running with Tellor, providing your project with a fully decentralized and censorship-resistant oracle. Overview Tellor is an oracle system where parties can request the value of an off-chain data point (e.g. BTC/USD) and reporters compete to add this value to an on-chain data-bank, accessible by all Polygon smart contracts. The inputs to this data-bank are secured by a network of staked reporters. Tellor utilizes crypto-economic incentive mechanisms. Honest data submissions by reporters are rewarded by the issuance of Tellorâ€™s token. Any bad actors are quickly punished and removed from the network by a dispute mechanism. In this tutorial we'll go over: - Setting up the initial toolkit you'll need to get up and running. - Walk through a simple example. - List out testnet addresses of networks you currently can test Tellor on. Using Tellor The first thing you'll want to do is install the basic tools necessary for using Tellor as your oracle. Use this package to install the Tellor User Contracts: npm install usingtellor Once installed this will allow your contracts to inherit the functions from the contract 'UsingTellor'. Great! Now that you've got the tools ready, let's go through a simple exercise where we retrieve the bitcoin price: BTC/USD example Inherit the UsingTellor contract, passing the Tellor address as a constructor argument: Here's an example: solidity import "usingtellor/contracts/UsingTellor.sol"; contract PriceContract is UsingTellor { uint256 public btcPrice; //This Contract now has access to all functions in UsingTellor constructor(address payable tellorAddress) UsingTellor(tellorAddress) public {} function setBtcPrice() public { bytes memory b = abi.encode("SpotPrice",abi.encode("btc","usd")); bytes32 queryID = keccak256(b); uint256 timestamp; bytes value; (value, timestamp) = getDataBefore(queryId, block.timestamp - 15 minutes); btcPrice = abi.decode(value,(uint256)); } } Addresses Tellor Tributes: 0xe3322702bedaaed36cddab233360b939775ae5f1 Oracle: 0xD9157453E2668B2fc45b7A803D3FEF3642430cC0 Testing Polygon Mumbai Testnet: 0xD9157453E2668B2fc45b7A803D3FEF3642430cC0 Test Tributes: 0xCE4e32fE9D894f8185271Aa990D2dB425DF3E6bE Need some test tokens? Tweet us at '@trbfaucet' For ease of use, the UsingTellor repo comes with a version of the Tellor Playground contract for easier integration. See here for a list of helpful functions. !!! info For a more robust implementation of the Tellor oracle, check out the full list of available functions here. !!! info Still have questions? Join the community here! # umbrella.md: !!! info "Content disclaimer" Please view the third-party content disclaimer here. Umbrella Network is a decentralized oracle service that provides blockchain projects with secure, scalable, and customizable data solutions. The service leverages a network of decentralized community validators to ensure the reliability of the oracle data. Solutions Umbrella Network provides three data consumption options: on-chain data, layer 2 data, and on-demand data. On-chain data Retrieve data feeds directly on-chain. Projects can define parameters such as deviation triggers and heartbeat to manage the frequency and precision of the updates. This is the simplest and most straightforward approach for reading data. Layer-2 data Read data feeds off-chain and verify them on-chain to ensure secure utilization within your dApp. Prices are regularly updated off-chain at a predefined frequency, and web3 apps fetch this data when needed and submit it on-chain. This approach is particularly advantageous when your project necessitates handling large volumes of data, as storing it entirely on-chain would be cost-prohibitive. On-demand data On-demand data fetching planned for Q3 2023. Resources Technical documentation The official technical documentation can be found here. Custom feeds In case your project requires a custom feed, please contact the Umbrella team here. # crusthelpers.md: !!! caution "Content disclaimer" Please view the third-party content disclaimer here. Crust Network provides a decentralized storage network for Web3. It is designed to realize the core values of decentralization, privacy and assurance. Crust supports multiple storage-layer protocols such as IPFS, and exposes instantly accessible on-chain storage functions to users. The protocol matches people who have hard drive space to spare with those who need to store data or host content. Crust supports most smart contract platforms with its cross-chain solution, including Polygon. Find out all the platforms that Crust storage supports on their official website. !!! info "Learn more about Crust Network" Check out the Decentralized Storage Market and Cross-chain Storage Solution offered by Crust Network. Also, you can start building with Crust Build-101. Crust storage helpers - Crust Files: Built by the Crust community, Crust Files is the first personal cloud storage for Web3 that supports multi-web3 wallet login and provides fully decentralized and privacy storage function for all users. - IPFS W3Auth Gateway: A Web3 authenticated IPFS gateway that supports all the IPFS WRITE functions and helps users upload files or host content after passing the W3Auth verification. - IPFS W3Auth Pinning Service: A standard IPFS pinning service with Web3 authentication that allows users to access Crust storage resources without needing to interact with the Crust blockchain. - Crust Github Action: A standard Github action which can include fully decentralized CI/CD hosting with Crust. # filecoinhelpers.md: !!! caution "Content disclaimer" Please view the third-party content disclaimer here. Filecoin is built on top of IPFS and supports storing data long-term via on-chain deals. Together, they help us break free from centralized services while conveniently allowing us to enjoy the same luxuries of speed and guaranteed storage that centralized services would bring. Storage helpers (IPFS + Filecoin) - Spheron.Network: Deploy, manage & scale your applications utilising Web3 Infrastructure. (video). - Lighthouse.storage: Permanent storage on IPFS and Filecoin that allows users to pay once and store forever. Lighthouse can also be used to store private data and build token gated applications on Filecoin. - Web3.storage: Data storage service that stores and retrieves data on IPFS and Filecoin (video). - NFT.Storage Flagship Product: NFT storage service that stores and retrieves data relating to NFTs on IPFS and Filecoin for a one-time fee charged per GB of storage. - NFT.Storage Classic (uploads decommissioned): Used to offer free storage for NFT data. While you can't upload new data anymore, any files uploaded before June 30, 2024 remain accessible. # ipfs.md: !!! caution "Content disclaimer" Please view the third-party content disclaimer here. Context Polygon blockchain reduces transaction costs to store data versus Ethereum mainnet; however, even these lower costs add up quickly when storing sizable files. Developers also must consider block size constraints and transaction speed limitations when storing data onchain. One solution which addresses all of these concerns is IPFS, the InterPlanetary File System. What is IPFS? IPFS is a distributed system for storing and accessing files, websites, applications, and data. IPFS uses decentralization, content addressing, and a robust peer-to-peer network of active participants to allow users to store, request, and transfer verifiable data with each other. Decentralization makes it possible to download a file from many locations that aren't managed by one organization, providing resilience and censorship resistance right out of the box. Content addressing uses cryptography to create a uniquely verifiable hash based upon what is in a file rather than where it is located. The resulting content identifier (CID) provides assurance a piece of data is identical regardless of where it is stored. Finally, an ever growing active community of users makes this peer-to-peer sharing of content possible. Developers upload and pin content to IPFS while Filecoin or Crust storage providers help to ensure persistent storage of that content. IPFS based storage allows you to simply store the CID for your content rather than loading entire files to Polygon blockchain; allowing for decreased costs, larger file sizes, and provably persistent storage. For more details refer IPFS Docs. Example projects 1. Tutorial in scaffold-eth that demonstrates how to mint an NFT on Polygon with IPFS - link. 2. Building a full stack web3 app with Next.js, Polygon, Solidity, The Graph, IPFS, and Hardhat - link. # getting-started.md: !!! warning "Content disclaimer" Please view the third-party content disclaimer here. This page serves as a resource for using Polygon compatible wallets which have key management and interfaces that allow users to perform chain actions and sign transactions, such as sending assets. If you are new to the concepts and terminology of wallets, start at the wallet basics section of this page. You can browse through the many third-party wallets that support Polygon chains or, if you already have Polygon networks in your wallet of choice, check out Polygon's native asset management solution Polygon Portal. !!! info "Centralized Exchanges (CEXs)" For a list of CEXs that support Polygon, visit a third-party tracking website such as CoinGecko. Wallet basics Comprehensive information and guides on wallets, including what a wallet is, how to use it, types of wallets, and key terminology can be found on Ethereum.org's wallet documentation page. Important wallet safety information You are responsible for keeping your keys safe and secure. There are many scams and pitfalls in managing your wallet and protecting it from malicious situations where bad actors try to compromise your wallet. Ethereum.org documentation has a how to stay safe section on their wallet documentation page (scroll to the middle of the page) referenced in the screenshot below. It includes some tips and links to more comprehensive articles. `![Ethereum.org: how to stay safe](https://ethereum.org/en/wallets/)` There are companion plugins for wallets that help detect, warn, and prevent malicious wallet scams and transactions. Visit the wallet security tools section on the Alchemy website. Look for the Wallet Security Tools tag for tools you can use to protect your wallet. !!! warning "Wallet safety" The official Polygon Support cannot provide assistance for stolen wallet assets and third-party wallet security plugins/software. Do your own research and be cautious. Third party wallets !!! info "Third-party" Some third party wallets may not have Polygon networks natively as an option in their blockchain network selection. It is easy to fix this by using Chainlist. Search for the Polygon network you want to add to your wallet, and click the Connect Wallet button. More information and an example of adding a mainnet or testnet Polygon chain using Chainlist MetaMask wallet can be found here. !!! warning "Third-party wallets" Do your own due diligence before using third party wallets. The official Polygon Support cannot provide assistance for issues with these wallets or other non-native wallets. Ethereum.org's wallet documentation page has a comprehensive page of different wallets you can select from. There are many options to filter the list to find a wallet that is right for you including: - Device type: browser plugin, desktop, mobile, or hardware wallet. - Security: open source, personal ownership/custody of your wallet. - Language support: filter by your preferred language. Be sure to select Layer 2 on the Filter menu on the left side of the page to list wallets that are compatible with Polygon networks.

`![ethereum.org Wallet List](https://ethereum.org/en/wallets/find-wallet/)`

Polygon portal The Polygon Portal is a comprehensive solution for account management on Polygon. The Portal can be used to perform a variety of tasks, such as: - Bridge your assets via the Polygon native bridge and a variety of third-party bridges. - View your assets and token lists. - Use the refuel gas feature to purchase MATIC or ETH for gas on the destination chain. - Swap assets with third-party DEXs. For more details on what features Polygon Portal offers, and how-to guides, visit the Polygon Portal documentation page. # openfort.md: !!! caution "Content disclaimer" Please view the third-party content disclaimer here. Openfort is an infrastructure provider designed to simplify the development of games and gamified experiences across their suite of API endpoints. The platform vertically integrates the AA stack, offering three core components: Identity, Cloud, and Ecosystems. Account abstraction infrastructure built for web3 games Identity - Authentication: Openfort's authentication service offers a comprehensive solution for user onboarding in web3 gaming by integrating with various backend solutions and authentication platforms. - Private Key Management: Utilizing an MPC SSS solution, Openfort ensures secure and efficient private key handling. - Smart Contract Wallets: Accounts are depicted as on-chain entities that secure a user's assets, acting as digital lockers. User accounts involve interactions with smart accounts, providing a secure and flexible account model. Cloud - Transaction Management: Openfort's transaction system is designed to handle transactions at scale, supporting high throughput for games. - Gasless Transactions: Simplify user experience by abstracting away gas fees from end-users. - Parallelization: Efficient processing of multiple transactions simultaneously to enhance performance. - Nonce Management: Automatic handling of transaction nonces to prevent conflicts and ensure smooth execution. Ecosystems - Middleware Deployment: Deploy Openfort's middleware technology in your chain from day zero, enabling quick integration and scalability. - Whitelabel Onboarding: Customize the onboarding experience to match your ecosystem's branding and requirements. - Ecosystem Wallets: Tailored wallet solutions that integrate seamlessly with your specific ecosystem. - Ecosystem Policies: Implement and manage policies specific to your ecosystem, ensuring compliance and optimal functionality. Get started by forking live samples of Openfort infrastructure to explore these components in action. # particle-network.md: !!! caution "Content disclaimer" Please view the third-party content disclaimer here. Particle Network's Wallet Abstraction SDKs facilitate 2-click user onboarding into EOAs/SCAs through passkeys, social logins, or typical Web3 wallets. Developers can integrate Particle's Wallet Abstraction suite via APIs and SDKs for mobile and desktop, acting as an all-in-one method of bringing users into your Polygon application, regardless of their background (Web3 natives, brand new users, and so on). Additionally, across various EVM chains, including Polygon, Particle's SDKs can facilitate full-stack, modular implementation of ERC-4337 Account Abstraction. Particle Wallet itself is available either in an application-embedded format, depending on the type of integration a specific developer chooses, or standalone through the mobile or web application, and it can be integrated via various SDKs. This page will cover Particle Connect. Particle Connect is a React-based SDK that offers a unified solution for managing user onboarding through social logins (via Particle Auth) and standard Web3 wallets. This creates a consistent and accessible experience for Web3-native users and traditional consumers. - Type: Non-custodial. - Private Key Storage: Userâ€™s local device/encrypted and stored with Particle. - Communication to Ethereum Ledger: Mixed/Particle. - Key management mechanism: MPC-TSS. Integrating Particle Connect The Particle Connect SDK is the primary tool for facilitating wallet creation, login, and interaction with Particle. It provides a unified modal for connecting through social logins (via Particle Auth) or traditional Web3 wallets, ensuring an accessible experience for both Web3 users and mainstream consumers. Install dependencies js yarn add @particle-network/connectkit viem@^2 OR js npm install @particle-network/connectkit viem@^2 Configure Particle Connect Now that you've installed the initial dependencies, you'll need to head over to the Particle Network dashboard to create a project & application so that you can acquire the required keys/IDs (projectId, clientKey, and appId) for configuration. After obtaining your project keys, you can configure the SDK by wrapping your application with the ParticleConnectkit component. This allows you to apply customizations and input the project keys. Here is an example of a Connectkit.tsx file (based on Next.js) exporting the ParticleConnectkit component: ts "use client"; import React from "react"; import { ConnectKitProvider, createConfig } from "@particle-network/connectkit"; import { authWalletConnectors } from "@particle-network/connectkit/auth"; import { polygon, polygonAmoy } from "@particle-network/connectkit/chains"; const config = createConfig({ projectId: process.env.NEXTPUBLICPROJECTID!, clientKey: process.env.NEXTPUBLICCLIENTKEY!, appId: process.env.NEXTPUBLICAPPID!, walletConnectors: [authWalletConnectors({})], chains: [polygon, polygonAmoy], }); // Wrap your application with this component. export const ParticleConnectkit = ({ children }: React.PropsWithChildren) => { return {children}; }; !!! note "" This is a minimal version of the ConnectKit.tsx file, providing just enough to get started. For complete details and full implementation guidelines, refer to the Particle Network documentation. Then, wrap your application with the ParticleConnectkit component. Here is an example of a layout.tsx file: ts import { ParticleConnectkit } from '@/connectkit'; import type { Metadata } from 'next'; import { Inter } from 'next/font/google'; import './globals.css'; const inter = Inter({ subsets: ['latin'] }); export const metadata: Metadata = { title: 'Particle Connectkit App', description: 'Generated by create next app', }; export default function RootLayout({ children, }: Readonly<{ children: React.ReactNode; }>) { return ({children}); } Facilitating login/connection With Particle Connect configured, you can enable social logins in your application using the ConnectButton component. ts import { ConnectButton, useAccount } from '@particle-network/connectkit'; export const App = () => { const { address, isConnected, chainId } = useAccount(); // Standard ConnectButton utilization return (

{isConnected && (<>

Address: {address}

Chain ID: {chainId}

)}

); }; For managing interactions at the application level after onboarding, @particle-network/connectkit offers various hooks. You can explore all the available hooks in the Particle Connect SDK documentation. Particle Connect Quickstart Explore the Particle Connect Quickstart in the Particle Network documentation for a step-by-step guide on starting and configuring a new project. # portal.md: Polygon Portal is an integrated UI platform that serves as a one-stop solution for asset management and token operations such as swapping and bridging. The unified UI comes with a comprehensive dashboard that can be used by connecting your wallet via popular tools such as MetaMask, Coinbase, Bitski, Venly, WalletConnect, and more. The portal establishes a trustless, two-way transaction channel between the Polygon PoS and zkEVM, and Ethereum networks, along with support for LxLy chains in the Polygon Ecosystem. You can transfer your tokens across these chains without incurring third-party risks and market liquidity limitations. !!! info - The bridge is operational on both testnet and mainnet. - The native bridge currently does not support bridging tokens between Polygon PoS and zkEVM chains. Features - Faster transactions: The bridge integrates the Socket Widget to enable faster transactions. - Support for Kyberswap: You can swap your tokens via Kyberswap directly through the portal. - Support for third-party bridges and DEXs: The portal supports bridging and swapping tokens using third-party bridges natively through the same UI. - Refueling: The portal integrates the 0x API to enable seamless bridging for your tokens to Polygon chains and eliminates the need to swap your tokens for the respective native token (i.e., MATIC or ETH depending upon the destination chain), or purchase them separately, to pay for gas fees. !!! info The Refuel feature is limited to select tokens. Getting started In this tutorial, we'll cover how to: - Connect to Polygon Portal - Transfer funds from Ethereum to Polygon chains - Claim bridged tokens on Ethereum - Add custom tokens to Polygon Portal !!! info If you come across any issues, Polygon support is just a click away! Feel free to get in touch with us for assistance. Connect to Polygon Portal 1. Use the following URL to access Polygon Portal: https://portal.polygon.technology/ 2. Click on the Connect Wallet button to bring up the blocknative wallet connect window.

`![](../../img/tools/portal/portal-1.png){ width=60% }`

3. Select your wallet from the options available. We'll proceed with MetaMask for the scope of this tutorial. You can connect to the portal with a Coinbase, OKX, or another wallet that supports WalletConnect.

`![](../../img/tools/portal/portal-2.png){ width=60% } ![](../../img/tools/portal/portal-24.png){ width=60% }`

4. Once you approve the connection in your wallet, you'll be able to you asset balances across Ethereum and Polygon chains.

`![](../../img/tools/portal/portal-25.png){ width=60% }`

Transfer funds from Ethereum to Polygon chains 1. Start by navigating to the Polygon Portal, and then either selecting Bridge your Asset option on the main page, or the Bridge option from the sidebar on your dashboard.

`![](../../img/tools/portal/portal-3.png){ width=60% }`

2. Once you see the bridge widget on your main screen, input the source and destination chains.For instance, if you were transferring tokens from Ethereum to Polygon zkEVM, select the following options.

`![](../../img/tools/portal/portal-4.png){ width=60% }`

3. Select the token from the drop-down list that you're looking to bridge from the source chain to the destination chain.

`![](../../img/tools/portal/portal-5.png){ width=60% } ![](../../img/tools/portal/portal-6.png){ width=60% }`

4. Enter the transfer amount for your selected token either manually, or by selecting a percentage of your current balance. !!! tip "Bridging tokens to a different address" You'll also see the option to transfer the tokens to a different address that you can select to enter a wallet address different from the one connected to the portal. 5. At this point, you should be able to send the transaction through. Confirm the transaction in your wallet, and you'll be able to see an ETA for the funds to hit your wallet on the destination chain.

`![](../../img/tools/portal/portal-7.png){ width=60% }`

6. You can check the transaction status and browse your transaction history by selecting Transactions from the sidebar. Select a transaction to view details.

`![](../../img/tools/portal/portal-8.png){ width=60% } ![](../../img/tools/portal/portal-9.png){ width=60% }`

Claim bridged tokens on Ethereum MATIC tokens 1. Let's say you bridged MATIC from PoS to Ethereum. Once your tokens are ready to claim, you'll be able to select the Initiate Claim option under the transaction record.

`![](../../img/tools/portal/portal-14.png){ width=60% }`

2. Approve the transaction in your wallet, and then select "Claim Tokens".

`![](../../img/tools/portal/portal-15.png){ width=60% }`

3. The funds will be reflected in your wallet soon. You can keep an eye on the transaction status from this page. ETH and custom tokens 1. Your tokens transferred from Polygon zkEVM to Ethereum will generally be

available to claim in 30-45 minutes. You can track the transaction status on the "Transactions" page.
`![](../../img/tools/portal/portal-10.png){ width=60% }`
2. The Claim Tokens button will appear under the transaction once your tokens are ready to be claimed.
`![](../../img/tools/portal/portal-11.png){ width=60% }`
3. Upon claiming your tokens, you'll receive a prompt in your wallet to sign the transaction.
`![](../../img/tools/portal/portal-12.png){ width=60% }`
4. Once you sign the transaction successfully, the bridged tokens will be sent to your wallet, which your wallet balance will reflect shortly.
`![](../../img/tools/portal/portal-13.png){ width=60% }`
Add custom tokens to Polygon Portal 1. First, try looking up the token directly by clicking on the Manage Tokens located in the bottom-right corner of the dashboard, and then entering the token name in the input field.
`![](../../img/tools/portal/portal-16.png){ width=60% } ![](../../img/tools/portal/portal-26.png){ width=60% }`
2. If the token you're looking for shows up, select Add to add it to the portal. 3. If your token doesn't appear in the search results, you can also add the token manually by selecting the + in the bottom-right of the Manage tokens window.
`![](../../img/tools/portal/portal-20.png){ width=60% }`
4. Enter the token contract address in the input field.
`![](../../img/tools/portal/portal-21.png){ width=60% }`
5. If the token contract address is valid, you'll see the option at the bottom to add it to the portal. Select the option and the token will appear in the token list on the portal dashboard.
`![](../../img/tools/portal/portal-22.png){ width=60% } ![](../../img/tools/portal/portal-23.png){ width=60% }`

portis.md: !!! caution "Content disclaimer" Please view the third-party content disclaimer here. Portis is a web-based wallet built keeping easy user-onboarding in mind. It comes with a JavaScript SDK that integrates into the dApp and creates a local wallet-less experience for the user. Further, it handles setting up the wallet, transactions, and gas fees. Like MetaMask, it is non-custodial - users control their keys, Portis just stores them securely. But, unlike MetaMask, it is integrated into the application and not the browser. Users have their keys associated with their login ID and passwords. - Type: Non-custodial/HD. - Private Key Storage: Encrypted and stored on Portis servers. - Communication to Ethereum Ledger: Defined by the developer. - Private key encoding: Mnemonic. Set up web3 Install Portis in your dApp: sh npm install --save @portis/web3 Now, register your dApp with Portis to obtain a dApp ID using the Portis Dashboard. Import portis and web3 objects: js import Portis from '@portis/web3'; import Web3 from 'web3'; Portis constructor takes the first argument as the dApp ID and the second argument as the network you would like to connect with. This can either be a string or an object. js const portis = new Portis('YOURDAPPID', 'maticTestnet'); const web3 = new Web3(portis.provider); Set up account If the installation and instantiation of Web3 was successful, the following should successfully return the connected account: js this.web3.eth.getAccounts() .then((accounts) => { this.account = accounts[0]; }) Instantiating contracts This is how we should instantiate contracts: js const myContractInstance = new this.web3.eth.Contract(myContractAbi, myContractAddress) Calling functions Calling call() function js this.myContractInstance.methods.myMethod(myParams) .call() .then (// do stuff with returned values) Calling send() function js this.myContractInstance.methods.myMethod(myParams) .send({ from: this.account,gasPrice: 0 }) .then ((receipt) => { // returns a transaction receipt }) # torus.md: !!! caution "Content disclaimer" Please view the third-party content disclaimer here Torus is a user-friendly, secure, and non-custodial key management system for decentralized apps. We're focused on providing mainstream users a gateway to the decentralized ecosystem. - Type: Non-custodial/HD. - Private Key Storage: Userâ€™s local browser storage/encrypted and stored on Torus servers. - Communication to Ethereum Ledger: Infura. - Private key encoding: Mnemonic/social-Auth-login. Depending on your application needs, Torus can be integrated via the Torus Wallet, or by interacting directly with the Torus Network via CustomAuth. For more information, visit the Torus documentation. Torus wallet integration If your application is already compatible with MetaMask or any other Web3 providers, integrating the Torus Wallet would give you a provider to wrap the same Web3 interface. You can install via a npm package. For more ways and in-depth information, please visit the official Torus documentation on wallet integration. Installation sh npm i --save @toruslabs/torus-embed Example js title="torus-example.js" import Torus from "@toruslabs/torus-embed"; import Web3 from "web3"; const torus = new Torus({ buttonPosition: "top-left" // default: bottom-left }); await torus.init({ buildEnv: "production", // default: production enableLogging: true, // default: false network: { host: "mumbai", // default: mainnet chainId: 80001, // default: 1 networkName: "Mumbai Test Network" // default: Main Ethereum Network }, showTorusButton: false // default: true }); await torus.login(); // await torus.ethereum.enable() const web3 = new Web3(torus.provider); CustomAuth integration If you are looking to control your own UX, from login to every interaction, then you can use CustomAuth. You can integrate via one of their SDKs depending on the platform(s) you are building on. For more info, please visit Torus CustomAuth integration. # using-web3modal.md: !!!caution Content disclaimer Please view the third-party content disclaimer here. Overview Web3Modal is a simple and intuitive SDK that provides a drop-in UI to enable users of any wallet to seamlessly log in to applications, offering a unified and smooth experience. It features a streamlined wallet selection interface with automatic detection of various wallet types, including mobile, extension, desktop, and web app wallets. Code sandbox for Polygon The Web3Modal team has prepared a Polygon Code Sandbox. Itâ€™s a straightforward way for developers to integrate and get hands-on experience with Polygon. How to integrate 1. Visit Web3Modal: Go to Web3Modal's official website to explore its features and capabilities. 2. Explore the Code Sandbox: Utilize the Polygon Code Sandbox to demo and understand the integration process. 3. Follow the Documentation: Refer to the provided documentation and instructions to integrate Web3Modal into your projects and leverage its features effectively. zkEVM support If you need help with anything related to the Polygon zkEVM, you can raise a ticket on the Polygon Support portal, and check out the Knowledge base to view the most common queries about the zkEVM. Additionally, you can reach out to the support team available on the #zkevm-support channel on the Polygon Discord server. Instructions for raising a zkEVM support ticket are as follows: 1. Join the Polygon Discord server here. 2. Accept the invite sent via DM. 3. Take the Member role under #roles. 4. Navigate to the #zkevm-support channel. You can now contact the zkEVM support staff with your questions and concerns. We will actively monitor for issues and work to resolve them as soon as possible. # walletconnect.md: !!! caution "Content disclaimer" Please view the third-party content disclaimer here. WalletConnect is an open protocol - not a wallet - built to create a communication link between dApps and wallets. A wallet and an application supporting this protocol will enable a secure link through a shared key between any two peers. A connection is initiated by the dApp displaying a QR code with a standard WalletConnect URI and the connection is established when the wallet application approves the connection request. Further requests regarding funds transfer are confirmed on the wallet application itself. Set up web3 To set up your dApp to connect with a userâ€™s Polygon Wallet, you can use WalletConnectâ€™s provider to directly connect to Polygon. Install the following in your dApp: bash npm install --save @maticnetwork/walletconnect-provider Install matic.js for Polygon integration: bash npm install @maticnetwork/maticjs And add the following code in your dApp; js import WalletConnectProvider from "@maticnetwork/walletconnect-provider" import Web3 from "web3" import Matic from "maticjs" Next, set up Polygon and Sepolia provider via WalletConnectâ€™s object: javascript const maticProvider = new WalletConnectProvider({ host: https://rpc-amoy.polygon.technology, callbacks: { onConnect: console.log('connected'), onDisconnect: console.log('disconnected!') } }) const sepoliaProvider = new WalletConnectProvider({ host: https://ethereum-sepolia-rpc.publicnode.com, callbacks: { onConnect: console.log('connected'), onDisconnect: console.log('disconnected') } }) We created the above two provider objects to instantiate our Web3 object with: js const maticWeb3 = new Web3(maticProvider) const sepoliaWeb3 = new Web3(sepoliaProvider) Instantiating contracts Once we have our web3 object, the instantiating of contracts involves the same steps as for Metamask. Make sure you have your contract ABI and address already in place. js const myContractInstance = new this.maticWeb3.eth.Contract(myContractAbi, myContractAddress) Calling functions !!! info The private key will remain in the userâ€™s wallet and the app does not access it in any way. We have two types of functions in Ethereum, depending upon the interaction with the blockchain. We call() when we read data and send() when we write data. Calling call() functions Reading data doesnâ€™t require a signature, therefore the code should be like this: js this.myContractInstance.methods .myMethod(myParams) .call() .then (// do stuff with returned values) Calling send() functions Since writing to the blockchain requires a signature, we prompt the user on their wallet (that supports WalletConnect) to sign the transaction. This involves three steps: 1. Constructing a transaction 2. Getting a signature on the transaction 3. Sending signed transaction js const tx = { from: this.account, to: myContractAddress, gas: 800000, data: this.myContractInstance.methods.myMethod(myParams).encodeABI(), } The above code creates a transaction object which is then sent to userâ€™s wallet for signature: js maticWeb3.eth.signTransaction(tx) .then((result) =>{ maticWeb3.eth.sendSignedTransaction(result) .then((receipt) => console.log (receipt)) }) signTransaction() function prompts the user for their signature and sendSignedTransaction() sends the signed transaction (returns a transaction receipt on success). # walletkit.md: !!! caution "Content disclaimer" Please view the third-party content disclaimer here. !WalletKit WalletKit is an all-in-one platform for adding smart, gasless wallets to your app. WalletKit offers pre-built components for onboarding users with email and social logins, which can be integrated in under 15 minutes using their React SDK or the wagmi connector. Alternatively, build completely bespoke experiences for your users using WalletKit's Wallets API. WalletKit is compatible with most EVM chains, including Polygon. It has integrated support for ERC-4337 and comes with a paymaster and bundler included, requiring no extra setup. You can check out the WalletKit documentation here. Start building for free on the Polygon testnet today. Integration Install the SDK bash npm i @walletkit/react-link walletkit-js or bash yarn add @walletkit/react-link walletkit-js Setup WalletKit Initialize WalletKitLink with your Project ID and wrap your app with WalletKitProvider, adding it as close to the root as possible. You can get your Project ID from the API Keys page in the WalletKit dashboard. ts import { WalletKitLink, WalletKitLinkProvider } from "@walletkit/react-link"; const wkLink = new WalletKitLink({ projectId: "", }); export function App() { return ...; } > â„¹ï¸ If you'd like to integrate WalletKit with wagmi, check out the installation docs here. # add-polygon-network.md: !!! warning "Content disclaimer" Please view the third-party content disclaimer here. To track your assets and send transactions on any of the Polygon networks using MetaMask, you need to add the respective network configurations to the wallet. In this doc, we demonstrate a few ways to do this for Polygon PoS testnet (Amoy) and mainnet. You can use the same methods to add Polygon zkEVM to your MetaMask wallet. ChainList 1. Depending on the network profile that you want to add to your MetaMask wallet, use one of the following links to navigate to the respective ChainList page. - Polygon PoS testnet (Amoy) - Polygon PoS mainnet - Polygon zkEVM testnet (Cardona) - Polygon zkEVM mainnet 2. Select the Add to Metamask option on the page. This brings up your MetaMask wallet.
!chainlist-1{width=50%}
3. Select the Approve option. This lets ChainList add the network configuration such as the network RPC URL, the chain ID, etc., to your MetaMask wallet.
!chainlist-2{width=50%}
4. Finally, select Switch network to switch to Amoy testnet in MetaMask.
!chainlist-3{width=50%}
5. You can now see your MATIC balance on Amoy. You can also switch between Amoy and other networks directly from the drop-down menu in the top-left corner.
!chainlist-4{width=50%}
Polygonscan 1. Navigate to the Polygonscan website. 2. Select the network you want to add to your MetaMask wallet from the drop-down list in the top-right corner of the home page.
!polygonscan-1{width=50%}
3. The explorer window refreshes and loads the explorer home page for the network you selected. 4. Next, scroll down to the bottom of the page, and select the button in the bottom-left corner prompting you to add the network to your MetaMask wallet. For instance, in the case of Amoy testnet, the button says Add Polygon Amoy Network.
!polygonscan-2{width=50%}
5. Select Approve from the MetaMask window. This allows the explorer to add the network configuration to your wallet.
!polygonscan-3{width=50%}
6. Finally, click on Switch network to switch to your selected network.
!polygonscan-4{width=50%}
7. You can now see your MATIC balance on Amoy. You can also switch between Amoy and other networks directly from the drop-down menu in the top-left corner.
!polygonscan-5{width=50%}
Add a network manually MetaMask gives you the option to add a network profile manually. Follow the MetaMask guide to add a custom network. The following table contains the mainnet and testnet network configurations for Polygon PoS and zkEVM. | Network | RPC URL | Chain ID | Native token | Explorer URL | | :---------------------: | :-------------------------------------: | :------: | :----------: | :-----------------------------------: | | PoS mainnet | https://polygor mainnet.infura.io | 137 | MATIC | https://polygonscan.com/ | | PoS Amoy (testnet) | https://rpc-amoy.polygon.technology | 80002 | MATIC | https://amoy.polygonscan.com | | zkEVM mainnet | https://zkevm-rpc.com | 1101 | ETH | https://zkevm.polygonscan.com | | zkEVM Cardona (testnet) | https://etherscan.cardona.zkevm-rpc.com/ | 2442 | ETH | https://cardona-zkevm.polygonscan.com | # overview.md: !!! warning "Content disclaimer" Please view the third-party content disclaimer here. MetaMask is a crypto wallet for web browsers and mobile devices that interacts with the Ethereum blockchain. It allows you to run Ethereum dApps in your browser without running a full Ethereum node. - Type: Non-custodial/HD - Private key storage: User local browser storage - Ethereum connection: Infura - Private key encoding: Mnemonic !!! tip - Backup your Secret Recovery Phrase. - If your device breaks, is lost, stolen, or has data corruption, there is no other way to recover it. -The Secret Recovery Phrase is the only way to recover your MetaMask accounts. Check more Basic Safety and Security Tips for MetaMask. # create-wallet.md: !!! caution "Content disclaimer" Please view the third-party content disclaimer here. If you are looking for a user-friendly wallet on the Polygon network, consider creating a Venly wallet. It allows you to enable a recovery mechanism and comes with end-user support via their in-app chat. Sign up to Venly Step 1 → Navigate to https://wallet.venly.io/ !Sign up to Venly Step 2 → Create an Account if you are new to Venly. You can sign up to Venly with your social credentials or using your email and password. !Create an account Step 3 → After signing up with one of your social accounts or with your email and password, you will need to accept their terms and conditions. !img Step 4 → An authentication code will be sent to your email address. Enter the code and click Submit. !img Step 5 → To secure the wallet, you must configure a PIN. Your PIN should be 6 digits and it will be used to approve future transactions. After entering the PIN, press Set and it will take you to the Venly Wallet page. !img Step 6 → Next, it is mandatory to enable emergency code. This will help with PIN resets in case you forget your PIN. Click Next. !img Step 7 → Confirm your PIN and click Create. !img Step 8 → - Copy the emergency code and save it somewhere safe. (It will be used for PIN resets) - Check the box confirming you stored the emergency code safely. (You cannot view it again) - Click Close. !img Create your wallet Now that you've configured your PIN, you are all set to create a Polygon wallet. Select Polygon from the list of various blockchain networks listed on Venly. !Select the Polygon blockchain To create a new wallet, select Create New Wallet. In case if you already have one, select the Import Wallet option to retrieve your wallet. !Create a new wallet Once you've pressed the button, Venly will ask you to confirm the new wallet creation using your PIN (the one you configured a few steps before). After that, your wallet will be created and you will be taken to your dashboard. !Wallet Dashboard Congratulations! You have now created your Polygon wallet using Venly to manage your digital assets on the Polygon network. Resources - Use Venly in your Dapp - Venly Widget - Web3 Wallet Providers # index.md: !!! caution "Content disclaimer" Please view the third-party content disclaimer here. Venly allows you to easily integrate your app to the Polygon blockchain, whether you already have an app integrated with Web3 or are building a new app from scratch. Venly provides a smooth and delightful experience for you and your users on both web & mobile. Venly will help you interact with the Polygon network, create blockchain wallets, create different asset types such as fungible (ERC20), and non-fungible tokens (ERC721 and ERC1155), and interact with smart contracts. Next to a superior developer experience, you can give your users a user-friendly interface. Each application is unique and has different needs, providing different ways of interacting with Venly. We recommend that web3 applications integrate the Venly web3 provider, others may use the Venly Widget. Key features - Support web and mobile. - Offers social logins. - Offers fiat-on-ramp. - Supports both Polygon and Ethereum. - Supports NFTs (ERC721 and ERC1155) on Polygon. - Easy to integrate using Web3. - Built for a mainstream audience. - Offers in-app customer support. Getting started If you already support web3 technology, you can improve your application's UX by integrating the Venly web3 provider, a smart wrapper around the existing web3 Ethereum JavaScript API. By using our web3 provider, you can leverage the full potential of Venly with minimal effort, and you will be able to onboard less tech-savvy users without making them leave your application or download third-party plugins. Integrating just takes two steps and 5 minutes! Don't support web3 yet? > Don't worry we've got you covered with our ðŸ"¦ Venly - Widget. Step 1: Add the library to your project Install the library by downloading it to your project via NPM. javascript npm i @venly/web3-provider Alternatively, you could also include the library directly from a CDN. javascript javascript Step 2: Initialize the web3 provider Add the following lines of code to your project, it will load the Venly web3 provider. Simple javascript import Web3 from 'web3'; import { VenlyProvider } from '@venly/web3-provider'; const Venly = new VenlyProvider(); const options: VenlyProviderOptions = { clientId: 'YOURCLIENTID' }; const provider = await Venly.createProvider(options); const web3 = new Web3(provider); Advanced javascript import Web3 from 'web3'; import { VenlyProvider } from '@venly/web3-provider'; const Venly = new VenlyProvider(); const options = { clientId: 'YOURCLIENTID', environment: 'staging', //optional, production by default signMethod: 'POPUP', //optional, REDIRECT by default bearerTokenProvider: () => 'obtainedbearertoken', //optional, default undefined //optional: you can set an identity provider to be used when

authenticating authenticationOptions: { idpHint: 'google' }, secretType: 'ETHEREUM' //optional, ETHEREUM by default }; const provider = await Venly.createProvider(options); const web3 = new Web3(provider); You can fetch wallets, and sign transactions, and messages. Congratulations, your dapp now supports Venly. Ready to try out the Wallet-Widget? Click here to get started. Want to know more about what Venly has to offer? Check out the documentation. More about Venly Venly stands out because of its commitment to supporting not only their wallet users by explaining what gas is, or by helping them import an Ethereum wallet into Polygon, but also the developers that are using Venly to build new products. At Venly, we offer a diverse range of products spanning various categories, including Wallet Solutions, NFT Tools, and Marketplaces. Let's begin with a brief overview of these three product categories: - Wallet solutions: Empower your users by providing them with a wallet or seamlessly integrating the Venly wallet into your application. - NFT tools: Facilitate the creation and distribution of NFTs, along with fetching comprehensive NFT data and information. - Marketplaces: Build your own NFT marketplace or list your NFT collection on our existing marketplace. Even in their test environments, they add an in-app chat so that developers can directly communicate with the team behind the Venly platform. In case you want to learn more, check out their detailed product documentation. Resources - Venly widget - Web3 wallet providers - Web3.js # venly-widget.md: Venly Widget Venly Widget is a JavaScript SDK created to streamline everyday blockchain tasks. Its purpose is to enable functionalities otherwise restricted due to security implications, such as creating signatures. By encapsulating Venly's extensive capabilities within a user-friendly JavaScript layer, Venly Widget empowers developers and simplifies the development process. !!! tip "Please note" If you are new to Web3 and don't have experience with blockchain technologies, we recommend you use the Venly Widget natively for a better developer experience. Create Polygon wallets with Venly Widget Venly Wallet allows you to create and manage wallets on the Polygon network. You can send and receive MATIC, Polygon NFTs, and ERC20 tokens. Apart from this, the Venly Wallet also supports: - Multiple blockchains. - Native token swap functionality. - Signing messages (including EIP712 message). - Calling smart contracts. - Importing wallets. Look and feel As the Widget is a product that incorporates a user interface (UI), let's look at how some of the more regular flows would appear for an end user. NFT transfer The application prompts users to transfer an NFT from their wallet to a different destination in this flow. !Polygon NFT transfer Token transfer The application prompts the user to transfer a token from their wallet to a different destination in this flow. !MATIC token transfer Integration options Multiple integration options are available to incorporate the Venly Widget into your application. Here is a brief overview of some of these options: 1. Native integration with Venly SDK: This approach involves utilizing the Venly SDK directly within your application's codebase. It allows you to access the full functionality of the Venly Widget and customize its behavior according to your requirements. 2. Ethers.js integration: You can integrate the Venly Widget with your application using the popular Ethers.js library. This involves utilizing the Ethers.js API to interact with the Venly Widget and manage user wallets, transactions, and other blockchain-related operations. 3. Web3Modal integration (WalletConnect): Web3Modal is a library that simplifies connecting to different wallet providers using standard protocols like WalletConnect. Integrating Web3Modal and WalletConnect enables users to interact with the Venly Widget and connect their wallets to your application seamlessly. These are just a few integration options to incorporate the Venly Widget into your application. The choice of integration method depends on your specific requirements, preferences, and the existing infrastructure of your application. !Venly widget integration options When to choose what? If you want to build your wallet app for users to interact with, you should use the Wallet API. If you want to integrate an existing and complete wallet solution, you can use the Venly Widget. There are multiple ways to integrate it - natively or by using another library (which also uses the Venly Widget in the background): !Decision making flowchart for web3modal and widget | Integration type | Description | UI flexibility | Blockchains | | :--------------- | :--- | :--- | :------------------- | | Native | A JavaScript SDK that seamlessly integ with various API functionalities, empowering users to execute diverse blockchain operations effortlessly. | The Widget delivers pre-designed screens tailored explicitly for end users, offering a ready-to-use solution. These screens are not customizable, ensuring consistency and simplicity in the user experience. | All supported chains | | Ethers.js | A JavaScript library is used to interact with the EVM blockchains. It provides a wide range of functionality for developers to build decentralized applications | This integration ensures that the Widget is invoked when needed, allowing users to conveniently and securely perform the required actions within the context of your application. | Only EVM chains | | Wagmi | A collection of React Hooks containing everything to work with EVMs. | This integration ensures that the Widget is invoked when needed, allowing users to conveniently and securely perform the required actions within the context of your application. | Only EVM chains | | Web3-React | A JavaScript SDK based on ethers.js. | This integration ensures that the Widget is invoked when needed, allowing users to conveniently and securely perform the required actions within the context of your application. | Only EVM chains | | Web3Modal | Web3Modal is a library that simplifies the process of connecting to different wallet providers using standard protocols like WalletConnect | When users opt to log in with Venly, the modal will initiate the Venly Widget upon various user actions, facilitating seamless integration between your application and the Venly platform. | Only EVM chains | !!! success Ready to try out the Venly Widget? Click here to read the getting started guide. # wallet-api.md: !!! caution "Content disclaimer" Please view the third-party content disclaimer here. Wallet API The Wallet API allows developers to interact with blockchain networks and offer wallet functionality to their users without having to build everything from scratch. This can include features like account creation, transaction management, balance inquiries, and more. - Welcome your users with custom wallet branding. You can customize the user interface to your taste. - You are completely in charge of the wallet user experience to optimize user conversion. Get total freedom with regard to UX and asset management with our Wallet API. - You and your users have complete control over digital assets without any third-party interference. Securely manage wallets with complete autonomy and privacy. - In the event of loss of login credentials, you and your users can recover access to wallets with a security code or biometric verification. !Wallet-API to create Polygon wallets Key features | Features | Description | | :-------------------------- | :-- | | management | Developers can use the API to create, manage, and secure wallets for their users. | | Transaction services | The API can enable the initiation and monitoring of blockchain transactions. | | Token support | It may allow the handling of various tokens and assets on supported blockchain networks. | | Blockchain interactions | Developers can integrate functionalities like reading data from the blockchain or writing data to it, along with creating and interacting with smart contracts. | | Security features | The API might offer features to enhance the security of user funds and transactions. | | User experience enhancement | It can contribute to a smoother and more user-friendly interaction with blockchain applications. | | Multi-blockchain support | Venly supports multiple blockchain networks, allowing developers to offer wallets for different cryptocurrencies. | Prerequisites 1. You need a Venly business account. If you don't have one, click to register in our Developer Portal, or follow our Getting Started with Venly guide. 2. You need an active trial or paid subscription of the Wallet-API. You can start a 30-day free trial for the Wallet-API. 3. You need your client ID and client secret which can be obtained from the Portal. Creating a Polygon wallet Request Endpoint: reference https POST /api/wallets Header params | Parameter | Param type | Value | Description | | :--------------- | :--------- | :--------- | :-------------------------------------- | | Signing-Method | Header | id:value | id: This is the ID of the signing method. value: This is the value of the signing method. | Body params | Parameter | Param type | Description | Data type | Mandatory | | :----------------------- | :--------- | :--- | :-------- | :-------- | | secretType | Body | The blockchain o which to create the wallet | String | âœ… | | userId | Body | The ID of the user who you want to link this wallet to | String | âŒ | | pincode (Deprecated) | Body | The pin that will encrypt and decrypt the wallet | String | âŒ | Request body json { "secretType": "MATIC", "userId": "6a5a9020-e969-4d9a-ae4b-fcd91a75769d" } Response body !!! success "Wallet created" The wallet has been created and linked to the specified user (userId). json { "success": true, "result": { "id": "10f26e75-1f01-4c46-a2c1-551f886c6db6", "address": "0x1bC722F33E62019F1d9CBcc43531c81731023483", "walletType": "APIWALLET", "secretType": "MATIC", "createdAt": "2023-12-06T12:17:36.420854289", "archived": false, "description": "Self-driven Zebra", "primary": false, "hasCustomPin": false, "userId": "6a5a9020-e969-4d9a-ae4b-fcd91a75769d", "custodial": false, "balance": { "available": true, "secretType": "MATIC", "balance": 0, "gasBalance": 0, "symbol": "MATIC", "gasSymbol": "MATIC", "rawBalance": "0", "rawGasBalance": "0", "decimals": 18 } } } !!! success "Ready to try it out?" Click to read the getting started guide for Wallet API. # index.md: --- hide: - toc ---

Polygon zkEVM

Polygon zkEVM is a Layer 2 network of the Ethereum Virtual Machine (EVM), a zero-knowledge (ZK) rollup scaling solution.

[Polygon zkEVM overview]

[Take a global view of the Polygon zkEVM network.]

[Connect wallet]

[Connect your wallet to zkEVM mainnet or testnet.]

[zkEVM local node]

[Get started by setting up a local zkEVM node.]

[zkEVM versus EVM]

[Discover the key differences between zkEVM and EVM.]

[Deploy zkEVM]

[Get started by deploying zkEVM.]

overview.md: Polygon zkEVM is to Ethereum a Layer 2 network and a scalability solution utilizing zero-knowledge technology to provide validation and fast finality of off-chain transactions. Polygon zkEVM supports a majority of Ethereum EIPs, precompiles, and opcodes. Developers benefit from the seamless deployment of smart contracts, developer tools, and wallets that already work on Ethereum, but in an environment with significantly lower costs. Connect to the fully-audited Polygon zkEVM mainnet or its testnet (Cardona testnet) using the details in the table below. | Network | RPC URL | ChainID | Block explorer URL | Gas token | | ------- | ------------------------------ | ---------------- | ---------------- | ----- | | Polygon zkEVM | https://zkevm-rpc.com | 1101 | https://zkevm.polygonscan.com/ | ETH | | Cardona zkEVM testnet | https://rpc.cardona.zkevm-rpc.com | 2442 | https://cardona-zkevm.polygonscan.com/ | ETH | Protocol development highlights The Polygon zkEVM testnet launched with a complete ZK proving system and full transaction data availability in October 2022. The proving system uses a combination of eSTARK proofs and FRI, that are then compressed using FFLONK SNARKs to create the final ZK proof. Following the launch of the testnet, the code base for Polygon zkEVM underwent several security audits. These were among the first audits ever performed on a complete, in-production ZK proving system. After the audits, Polygon zkEVM Mainnet Beta launched in March 2023. Since then, the zkEVM network has had two major upgrades: Dragon Fruit (ForkID5), in September 2023, and Inca Berry (ForkID6), in November 2023. All updates and upgrades of both the mainnet and testnet can be found in the Historical data document. Security measures zkEVM's upgrades are on par with Ethereum's security standards as they involve deployment of the following contracts: - An admin multisig contract to avoid having one account controlling upgrades. - A timelock contract to give users sufficient time delay to withdraw before execution. - A transparent upgradeable proxy, from OpenZeppelinâ€™s libraries of audited and battle-tested contracts. The activation of the 10-day timelock for upgrading zkEVM's smart contracts on Ethereum requires approval by the network's Admin, a three-participant multisig that acts as a governance tool for the protocol. This is a Gnosis Safe with a 2/3 threshold. In the event of an emergency that puts user funds at risk, the network's Security Council may remove the 10-day timelock. In such an emergency, the network state stops advancing and bridge functionality is paused. The Security Council is an eight-participant multisig. This is a Gnosis Safe with a 6/8 threshold. Learn more about zkEVM upgradability. Design characteristics Polygon zkEVM was designed with security in mind. As an L2 solution, it inherits its security from Ethereum. Smart contracts are deployed to ensure that everyone who executes state changes does so appropriately, creates a proof that attests to the validity of each state change, and makes validity proofs available on-chain for verification. Development efforts aim at permissionless-ness, that is, allowing anyone with the zkEVM software to participate in the network. For instance, the network allows anyone to circumvent any transaction-censorship by triggering the force batchesÂ mechanism, or to avoid denial of validity-proving by activating theÂ force verificationÂ feature. The ultimate aim is to ensure that there is no censorship and that no one party can control the network. Since data availability is most crucial for decentralization, Polygon zkEVM posts all transaction data and validity proofs on Ethereum. This means every Polygon zkEVM user has sufficient data needed to rebuild the full state of a rollup. Efficiency and overall strategy As a scalability solution, efficiency is key to Polygon zkEVM. The network therefore utilizes several implementation strategies to maximize efficiency. A few of these strategies are listed below: 1. Deployment of the consensus contract, which incentivizes the aggregator for participating in the proof generation process. 2. Carry out all computations off-chain while keeping only the necessary data and ZK-proofs on-chain. 3. Implementation of the bridge smart contract is made efficient by using only Merkle roots of exit trees. 4. Utilization of specialized cryptographic primitives within the proving component, zkProver, to speed up computations and minimize proof sizes. This is seen in: Running a special zero-knowledge assembly language (zkASM) for interpretation of bytecode. Using zero-knowledge technology such as zk-STARKs for proving purposes; these proofs are very fast though they are big in size. Instead of publishing the sizeable zk-STARK proofs as validity proofs, a zk-SNARK is used to attest to the correctness of the zk-STARK proofs. Publishing zk-SNARKs as the validity proofs to state changes. These help in reducing gas costs from 5M to 350K (wei). The Polygon zkEVM network is therefore secure, efficient, comes with verifiable block data, and cost-effective. # index.md: As an Ethereum Layer 2 scaling solution, Polygon zkEVM gathers transactions into batches after executing them. Aggregated batches are then dispatched to Ethereum for verification and validation, all managed through smart contracts. This document aims to offer a comprehensive understanding of the protocol design while delineating the currently implemented configuration. The major components of Polygon zkEVM are: - Consensus contract, which was initially PolygonZkEVM.sol has now been upgraded to PolygonZkEVMEtrog.sol. - zkNode, consisting of the synchronizer, sequencer, aggregator, and RPC. - zkProver for generating verifiable proofs of correct transaction executions. - zkEVM bridge for cross-chain messaging and transferring assets. The skeletal architecture of Polygon zkEVM is shown below: !Skeletal Overview of zkEVM Consensus contract The earlier version, Polygon Hermez 1.0, was based on the Proof of Donation (PoD) consensus mechanism. PoD was basically a decentralized auction conducted automatically, with participants (coordinators) bidding a certain number of tokens in order to be selected for creating batches. The updated consensus contract is designed to build upon the insights gained from PoD in v1.0 and adds support for permissionless participation of multiple coordinators in producing L2 batches. In the PoD mechanism, economic incentives were structured to require validators to operate with high efficiency to remain competitive. The latest version of the zkEVM consensus contract (deployed on Layer 1) is modeled after the Proof of Efficiency. While the protocol design is intended to support permissionless participation of multiple coordinators to generate L2 batches, for security reasons and given the protocol's early development stage, only one coordinator (referred to as the Sequencer) is currently operational. â€‹ Implementation model â€‹ Unlike the PoD, the Consensus Contract employs a simpler technique and is favoured due to its greater efficiency in resolving the challenges involved in PoD. The strategic implementation of the contract-based consensus ensures that the network: - Maintains its permissionless feature to produce L2 batches. - Is highly efficient, a criterion which is key for the overall network performance. - Attains an acceptable degree of decentralization. - Is protected from malicious attacks, especially by validators. - Maintains a fair balance between overall validation effort and network value. !!!tip Good to Know Possibilities of coupling the Consensus Contract (previously called Proof of Efficiency or PoE) with a PoS (Proof of Stake) are currently being explored. A detailed description is published on the Ethereum Research website. On-chain data availability â€‹ A full zk-rollup schema requires on-chain publication of both the transaction data (which users need to reconstruct the full state) and the validity proofs (zero-knowledge proofs). However, given Ethereum's current framework, publishing callback data to L1 incurs high gas fees, complicating the decision between opting for a full zk-rollup or a hybrid configuration. Under a Hybrid schema, either of the following is possible: - Validium: Data is stored off-chain and only the validity proofs are published on-chain. - Volition: For some transactions, both the data and the validity proofs are published on-chain, while in others, only the proofs are stored on-chain. Unless, among other considerations, the proving module can be significantly accelerated to alleviate high costs for validators, a hybrid schema remains a viable option. â€‹ PolygonZkEVMEtrog.sol â€‹ The underlying protocol in zkEVM ensures that the state transitions are correct by employing a validity proof. In order to ensure adherence to a set of pre-determined rules for state transitions, the consensus contract deployed on L1, is utilized. â€‹ !!! info The consensus contract is currently deployed on both Ethereum mainnet and Cardona testnet. â€‹ A smart contract verifies the validity proofs to ensure that each transition is

completed correctly. State transition verification is carried out by employing zk-SNARK circuits. A system of this type necessitates two processes: transaction batching and transaction validation. Polygon zkEVM implements two participant network roles in order to carry out its procedures: A sequencer and an aggregator. Under this two-layer model: - The sequencer proposes transaction batches to the network upon executing them. That is, it rolls up transactions into batches and adds them to the consensus contract. â€‹ - The aggregator checks the validity of the transaction batches and provides validity proofs. The smart contract, therefore, makes two calls: - Firstly, to receive batches from the sequencer. - Secondly to aggregators, requesting batches to be validated. â€‹ !Simplified proof of efficiency !!!info Although the current protocol design allows for several sequencers and aggregators, to prioritize security and given the early stage of protocol development, only one sequencer and one aggregator are currently operational. These are henceforth referred to as the trusted sequencer and the trusted aggregator. zkNode zkNode is the essential software for running a zkEVM node. It is a client that the network requires to implement the synchronization and govern the roles of the participants, including the trusted sequencer and the trusted aggregator. The consensus smart contract imposes the following requirements on the trusted sequencer and the trusted aggregator. â€‹ Sequencer - The trusted sequencer must run the software necessary for a full zkEVM node. - The trusted sequencer should pay a fee in the form of POL tokens for the right to create and process batches. - The trusted sequencer is incentivized to propose valid batches (consisting of valid transactions) with the fees paid by the network users who initiated the transactions. Aggregator A trusted aggregator receives all the transaction information from the trusted sequencer and sends it to the prover, which in turn provides a zk-proof after performing some complex computations. A special verfier smart contract validates the validity proof. The trusted aggregator's task is to therefore: - Provide validity proofs for the L2 transactions proposed by the trusted sequencer. - Run zkEVM's zkNode software and the zkProver for generating zero-knowledge validity proofs. - For a given batch, the trusted aggregator earns the POL fees (paid by the trusted sequencer) for providing a validity proof. - In a decentralized setting, each trusted aggregator needs to declare its intention to validate transactions, and subsequently compete to produce validity proofs based on its own strategy. Other than the trusted sequencer and the trusted aggregator, participants in the Polygon zkEVM network can take part as nodes that seek to know the state of the network. The zkNode architecture is modular in nature. You can dig deeper into zkNode and its components here. Incentivization structure The two permissionless participants of the zkEVM network are the trusted sequencer and the trusted aggregator. Proper incentive structures have been devised to keep the zkEVM network fast and secure. Below is a summary of the fee structure for the trusted sequencer and the trusted aggregator: - Sequencer - Collects transactions from a designated pool DB and puts them in a batch. - Receives fees paid by users who submitted the transactions. - Pays L1 transaction fees + POL (depending on pending batches). - The POL tokens go to the trusted aggregator. - Profitable if: txs fees > L1 call + POL fee. - Aggregator - Processes transactions published by the trusted sequencer. - Builds zkProof. - Receives POL tokens from the trusted sequencer. - Static cost: L1 call cost + server cost (to build a proof). - Profitable if: POL fees > L1 call cost + server cost See the Effective gas price document for how users can estimate the gas price to sign transactions with. zkProver zkEVM employs zero-knowledge technology to generate validity proofs. It uses a zero-knowledge prover (zkProver), which is intended to run on a server and is being engineered to be compatible with most consumer hardware. The trusted aggregator uses the zkProver to validate batches and provide validity proofs. The zkProver consists of a main state machine executor, a collection of secondary state machines (some with their own executor), a STARK-proof builder, and a SNARK-proof builder. !Skeletal overview of zkProver In a nutshell, the zkEVM expresses state changes in polynomial form. As a result, the constraints that each state transition must fulfil are polynomial constraints or polynomial identities. In other words, all valid batches must satisfy specific polynomial constraints. Check out the detailed architecture of the zkProver here. zkEVM bridge The zkEVM bridge is a smart contract that lets users transfer their assets between two layers, LX and LY. The L1-L2 in zkEVM is a decentralized bridge for secure deposits and withdrawal of assets. It is a combination of two smart contracts, one deployed on one chain and the second on the other. The L1 and L2 contracts in zkEVM are identical except for where each is deployed. Bridge L1 contract is on the Ethereum mainnet in order to manage asset transfers between rollups, while bridge L2 contract operates in a designated rollup, managing asset transfers between mainnet and the rollup(s). Layer 2 interoperability allows a native mechanism to migrate assets between different L2 networks. This solution is embedded in the bridge smart contract. Verifier Verifier is a smart contract which is able to verify any zk-SNARK cryptographic proof. The SNARK verifier proves the validity of every transaction in the batch. It is the core entity in any zk-rollup architecture as it verifies the correctness of each proof, thus ensuring the integrity of state transitions. The verifier contract is currently deployed on the Ethereum mainnet and Cardona testnet. # bookmarks.md: This document describes what bookmarks are, and how they are used in client-server protocol. A stream is a sequence of entries, where these entries belong to some specific operation, and are identified by an entry number. In the generic sense, an entry can be any piece of data relevant to an application's context. They can be either events or bookmarks. Events are defined by the application. Stream clients provide relevant information about an event that must be streamed. In the case of the Polygon zkEVM, entries are actually L2 blocks, while operations are thought of as batches. So then, when the stream source triggers the $\texttt{StartAtomicOp()}$ function and the corresponding message is sent to the stream server, that message in the Polygon zkEVM context, amounts to an instruction to "begin a batch, and prepare to receive its related entries (i.e., blocks) that are about to follow." Again, in the Polygon zkEVM context, the message sent when the $\texttt{CommitAtomicOp()}$ function is called, is tantamout to saying: "close the batch with the last entry received." Note that without these two messages, the stream server has no means of knowing where a batch starts and ends. Similarly, in the client-server protocol, there is a need for a stream client to indicate to the stream server what entry the streaming should begin with. A bookmark is used for this purpose by the stream client. What is a bookmark? A bookmark is an entry in a stream, and therefore has an entry number, denoted by $\texttt{entryNumber}$. A bookmark is essentially a string of bytes. It links an $\texttt{entryNumber}$ to a string of bytes in a way that is meaningful to an application. Stream clients can request stream servers to start the stream from a particular bookmark using the $\texttt{StartBookmark()}$ method. A bookmark is a type of a stream entry, used as an identifier. It points to a specific position in the stream from which the stream server must begin the streaming. !Figure StartBookmark command In addition to the $\texttt{start()}$ and $\texttt{stop()}$ commands of the client-server protocol, a command called $\texttt{StartBookmark()}$ is added. In the similar way the $\texttt{start()}$ and $\texttt{stop()}$ commands are identified by $\texttt{1}$ and $\texttt{2}$, respectively, the $\texttt{StartBookmark()}$ command is identified with the number 4. The $\texttt{StartBookmark()}$command is as follows: $$ \begin{aligned} 1.\qquad &\mathtt{u64\ command\ =\ 4} \\ 2.\qquad &\mathtt{u64\ streamType} \qquad //\ \texttt{e.g. 1:}\ \mathtt{zkEVM\ Sequencer} \\ 3.\qquad &\mathtt{u32\ bookmarkLength} \qquad //\ \texttt{Length of fromBookmark}\\ 4.\qquad &\mathtt{u8[\ ]\ fromBookmark} \end{aligned} $$ As previously seen with other commands, $\texttt{streamType}$ refers to the network associated with the stream source. When calling the $\texttt{StartBookmark()}$ function, and the corresponding message is sent to the stream server, the stream client must provide a bookmark. That is, a string of bytes together with the length of the string, denoted by $\texttt{bookmarkLength}$. Example: Consider the Polygon zkEVM case, where entries are L2 blocks. As seen in the above diagram, a bookmark precedes events. In this context, the data of each block is preceded by a bookmark entry, and this bookmark entry contains among other things the block number. It is more meaningful to use the $\texttt{blockNumber}$ in the Polygon zkEVM context. That is, the $\texttt{blockNumber}$ of the block with which the streaming must begin as requested by the stream client. Bookmarks DB Although stream entries are recorded in the stream file, there's a database called Bookmarks DB, which is specifically for storing all bookmarks. The Bookmarks DB is a key-value database that maps a bookmark (used as a key) with its entry number (used as the value). The technology behind Bookmarks DB is LevelDB, which is an open-source, fast key-value storage library written by Google's developers. So, when a stream client requests the stream server to start streaming from a specific bookmark, it provides the bookmark as a string of bytes. The stream server performs a binary search in order to locate the bookmark in the Bookmarks DB, and fetches the corresponding $\texttt{entryNumber}$ from the Bookmarks DB. It then begins to stream entries, starting from the $\texttt{entryNumber}$ onwards. !Figure All commands made by stream clients return a response, called a $\texttt{Result}$ entry. The format of such a $\texttt{Result}$ entry is as follows: $$ \begin{aligned} 1.\qquad &\texttt{u8 packetType // 0xff:Result} \\ 2.\qquad &\texttt{u32 length // Total length of the entry} \\ 3.\qquad &\texttt{u32 errorNum // Error code (e.g. 0:OK)} \\ 4.\qquad &\texttt{u8[] errorStr} \end{aligned} $$ Whenever a stream client calls the $\texttt{start()}$ function, the stream server responds with the $\texttt{Result}$ entry first, followed by all the entries. In the client-server protocol, each message from the stream server has its identifier or packetType. As seen in the code above, the $\texttt{Result}$ entry has the identifier, $\texttt{0xff}$, at the protocol level. The $\texttt{Result}$ identifier is followed by the total length of the entry. Next is a code of an error message, $\texttt{errorStr}$. If everything goes well then a $\texttt{0}$ is sent, meaning $\texttt{OK}$. A string is also sent, which is in a more human readable text. An error is sent in the form of a code and a string, where the string is just an array of bytes, and each byte is an ASCII character. # client-server-messages.md: This document gives further details on the client-server protocol messages. The stream client-server protocol messages are $\texttt{Start}$, $\texttt{StartBookmark}$, $\texttt{Stop}$, $\texttt{Header}$, $\texttt{Entry}$ and $\texttt{Bookmark}$â€‹. - The $\texttt{Start}$â€‹ message is sent from the stream client to the stream server in order to request for a stream to be sent. If the stream client wants to receive all the stream file, a $\texttt{Start}$ message is sent with an $\mathtt{entryNumber}$ $= \texttt{0}$. If the stream client knows the entry number at which the stream should start, it sends a $\texttt{Start}$ message with that particular entry number. That is, $\mathtt{entryNumber} \not= \texttt{0}$. - $\texttt{StartBookmark}$ message is the type of message the stream client can send if the stream client does not know the entry number, but knows something more meaningful to the application, like a bookmark. In the case of the Polygon zkEVM, if the stream client wants to receive information from a certain L2 block number, then it provides the appropriate bookmark by sending a $\texttt{StartBookmark}$ message. Such a bookmark is actually a codification of the L2 block number. - $\texttt{Stop}$ message is a message the stream client can send to the stream server if it wants to stop receiving the stream. - $\texttt{Header}$ message can be sent if the stream client requests just the header of a particular entry. We discuss the information an entry contains, in more detail, later in this document. In summary, an entry has two parts: a part called a $\texttt{Header}$, and a part called the $\texttt{data}$. So, a $\texttt{Header}$ message is used to request for just the header, but not the full data of an entry. It's like asking for only the block header and not the entire L2 block, in the Polygon zkEVM case. - $\texttt{Entry}$ message can be sent by the stream client in order to request for a specific entry in the stream file. Thus, instead of requesting the stream server to start streaming from a particular entry onwards, only one entry is obtained by sending an $\texttt{Entry}$ message. - $\texttt{Bookmark}$ message is a message in which a stream client sends a bookmark to the stream server, and the stream server in turn tells the stream client what entry number is linked to the bookmark. The above six messages are all messages used in the client-server protocol. And they can be found here. # data-streamer-design.md: The data streamer is another component of the zkEVM infrastructure through which external nodes can access raw block data. Any trustless node connected to the zkEVM can be up-to-date with the current L2 state by using the data streamer instead of using the JSON RPC. Any archive node that needs to receive all information about blocks and their transactions with minimal delay can use the data streamer. The usual procedure when working with the JSON RPC is that, for example; - The user's network node makes requests for information to the zkNode, - The zkNode fetches this information from the internal database, and - The zkNode relays it back to the requesting node. Developer experience has shown that although the JSON RPC is perfect for small queries, it is not an efficient way to obtain huge amounts of data often required for fetching the entire L2 state. A data streamer is best suited for the purpose of serving raw block data to external nodes that need to keep an up-to-date L2 state, irrespective of the required amount of data. Design approach The design of the data streamer emulates the conventional L1 networks, where blocks are delivered to all nodes in the network via some peer-to-peer protocol, typically referred to as the gossip protocol. With this approach the zkNode does not serve processed data via the JSON-RPC API to downstream nodes, but just â€fast streamsâ€ L2 data (which includes; batch data, block header data and transactions data) using a protocol with a low overhead. Instead of using existing streaming protocols, such as MQTT from IoT, the Polygon zkEVM team has developed a new custom-ware protocol, tailor-made for the zkEVM. Developing their own protocol has the advantage of owning the code, and thus the team can alter it as required for the purpose of improving the zkEVM network. The data streamer is a service that does not process blocks but sends them raw as they are generated. Data streamer building blocks We present a general abstraction of the data streamer. The basic architecture consists of a data source, data server, and data client, with an added option of a stream relay for better scalability. !Figure: Data streamer archtectural overview In the case of the Polygon zkEVM network, the data streamer is used by the sequencer. The stream source is in fact the zkNode, in particular, the sequencer zkNode. And stream clients are requesters of network data. By implementing the data streamer, it means stream clients do not have direct access to the stream source but go through stream servers. So, the data source sends L2 data to data servers, and data clients fetch the data from the stream servers. This way, there can be a few stream servers connected to the stream source, and many stream clients fetching L2 data from any of the stream servers. Ultimately, the stream server helps in easing congestion that could occur at the stream source. There are two protocols to consider in this framework; the server-source protocol, and client-server protocol. Stream relay For increased scalability, an intermediate node called stream relay is added to the architecture. And it is utilized as a "relayer" node between a stream server and a number of stream clients. Such a stream relay is a client to some stream server but acts as a server to connected stream clients. That is, each connected stream client sends messages to the stream relay according to the afore-mentioned client-server protocol. And, the stream relay in turn acts as a client to a stream server, to which the stream relay sends the messages received from stream clients, also in line with the client-server protocol. This design creates a highly scalable data streaming infrastructure. !Figure # data-streamer-protocols.md: This document explains the two data streamer protocols; server-source protocol, and client-server protocol. Server-source protocol The stream source and stream servers connect via TCP, which is the most reliable way to send and receive data over the internet. The server-source protocol, built on top of the TCP connection, involves transmission of generic messages between the stream source and a stream server. In a generic sense, the messages being sent from the stream source to the stream server are entries related to a particular atomic operation. An operation is atomic if either all the entries of the operation are included in the flow and streamed, or none of them are streamed. The protocol starts with the stream source calling the $\texttt{StartAtomicOp()}$ function, which sends the corresponding message to the stream server, indicating that the stream source is ready to send entries associated with a particular atomic operation $\texttt{Op}$. !Figure The number of the sent entries depends on a particular atomic operation $\texttt{Op}$. Entries received in the stream server are sequentially numbered across all atomic operations, and are stored in a file called stream file. For instance, if there are two atomic operations $\mathtt{OpA}$ and $\mathtt{OpB}$, where $\mathtt{OpA}$ has four (4) entries and $\mathtt{OpB}$ has three (3) entries, then the entries are sequenced as follows: $$ \texttt{entry}\ \mathtt{6}\ ||\ \texttt{entry}\ \mathtt{5}\ ||\ \texttt{entry}\ \mathtt{4}\ ||\ \texttt{entry}\ \mathtt{3}\ ||\ \texttt{entry}\ \mathtt{2}\ ||\ \texttt{entry}\ \mathtt{1}\ ||\ \texttt{entry}\ \mathtt{0} $$ When all the entries have been sent, the stream source calls a function called $\texttt{CommitAtomicOp()}$ which automatically sends a commit message to the stream server. Upon receipt of the commit message, the stream server can begin sending the entries to stream clients. Entries are streamed sequentially by the stream-server on a "first-received-first-sent" basis. There's a possibility to rollback the atomic operation using the $\texttt{RollbackAtomicOp()}$ function, and thus stopping all entries related to atomic operations from being streamed to stream clients. However, a rollback of an atomic operation can only be executed if the operation has not been committed. That is, an operation can only be rolled if its corresponding $\texttt{CommitAtomicOp()}$ function has not been called, and hence a commit message has not been sent to the stream server. The Polygon zkEVM team provides a library written in Go which allows the stream source to interact with stream servers. Client-server protocol The stream client and stream server also connect via TCP. In this protocol a stream client sends messages to a connected stream server, that are requests to send entries related to atomic operations. The basic protocol involves the stream client calling two main functions; $\texttt{start()}$ or $\texttt{stop()}$, and thus sending a corresponding message to the stream server in order to either start or stop sending a stream of entries. !Figure The start() message The stream client calls the $\texttt{start()}$ function which sends a message to the server, requesting data to be streamed. It can also request for the stream to begin at a specified entry number. The stream client's $\texttt{start()}$ function specifies three 64-bit unsigned integers, as follows: $$ \begin{aligned} 1.\qquad &\mathtt{u64\ command} = 1 \\ 2.\qquad &\mathtt{u64\ streamType} \qquad // \ \ \mathtt{e.g.,\ 1 :\ zkEVM sequencer} \\ 3.\qquad &\mathtt{u64\ fromEntryNumber} \end{aligned} $$ The value assigned to $\texttt{command}$ indicates whether a $\texttt{start()}$ or $\texttt{stop()}$ function has called by the client. As shown in the first line of the above code, $\mathtt{command\ =\ 1}$ means the stream client called the $\texttt{start()}$ function. In the second line, $\texttt{streamType}$ refers to the stream source node to which the stream server should connect. For example, $\mathtt{streamType\ =\ 1}$ means the source node is the zkEVM sequencer node. The third line specifies which entry the stream should start at. If $\mathtt{streamType\ =\ 1}$ and $\mathtt{fromEntryNumber\ =\ 74}$, then the stream starts at $\mathtt{blockNum\ =\ 74}$. Note that calling a $\texttt{start()}$ function in the middle of an active stream (i.e., while receiving entries) results in termination of the stream, and the TCP connection is also lost. Basically, receiving a "start-a-stream" message from the same stream client to which streaming is in progress, is interpreted by the stream server as an error. The stop() message The stream client can send another message to the stream server, by calling the $\texttt{stop()}$ function. Such a message is a request to stop streaming data. The stream client's $\texttt{stop()}$ function takes two 64-bit parameters, as shown below: $$ \begin{aligned} 1.\qquad &\mathtt{u64\ command\ = 2\ } \\ 2.\qquad &\mathtt{ u64\ streamType} \qquad //\ \mathtt{e.g.,\ 1 :\ zkEVM sequencer} \end{aligned} $$ Setting the value of $\mathtt{command}$ to $\mathtt{2}$, as shown in the first line of the above code, means the stream server should stop the stream. As in the case of the $\texttt{start()}$ function, the variable $\texttt{streamType}$ in the $\texttt{stop()}$ function codes, indicates the stream source to which the stream server must connect. The $\texttt{stop()}$ function does not specify any $\texttt{fromEntryNumber}$. Note that calling the $\texttt{stop()}$ function while streaming is in progress causes the stream to be terminated, but without losing the TCP connection. However, if the stream client triggers the $\texttt{stop()}$ function when there is no streaming in progress, the TCP connection terminates. # how-rollbacks-work.md: Recall that the server-source protocol begins with calling the $\texttt{StartAtomicOp}(\ )$, corresponding to which a message is sent to the stream server, preparing to receive entries related to a specific atomic operation. When the stream source sends the entries, the stream server appends the data of the entries to the stream file. Once all entries have been sent, the stream source calls the $\texttt{CommitAtomicOp}(\ )$ function, and the header of the stream file is subsequently updated. In particular, the $\texttt{totalLength}$ and $\texttt{totalEntries}$ fields. But if the $\texttt{RollbackAtomicOp}(\ )$ is triggered instead of the $\texttt{CommitAtomicOp}(\ )$, the header is not updated. In other words, the header of the stream file is updated only when the $\texttt{CommitAtomicOp}(\ )$ function is called. So, although some entries related to the atomic operation have already been added to the stream file, the header of the stream file is updated only with information of entries related to committed atomic operations. Since the $\texttt{RollbackAtomicOp}(\ )$â€‹ function can only be executed before a given atomic operation is committed, the header is not updated because the added entries (of the uncommitted atomic operation) will be overwritten with entries of the next atomic operation(s). This means a rollback amounts to overwriting entries in the stream file that are related to an uncommitted atomic operation. Example (Commit and rollback) Suppose an operation $\mathtt{OpA}$ has been started, and $\texttt{100}$ entries had already been added to the stream file when a rollback was triggered. Since a rollback was triggered before $\mathtt{OpA}$ was committed, the header of the stream file remains unaffected by the $\texttt{100}$â€‹ added entries. Let's say the $\texttt{totalLength}$ of the stream file is $\texttt{1712}$ when the rollback occurred. Although the actual length of the stream is $\texttt{1812}$, the $\texttt{totalLength}$ in the header of the stream file remains unchanged. Suppose that the next atomic operation $\mathtt{OpB}$ gets started and committed, but has only 40â€‹ related entries. Since $\mathtt{OpB}$ is committed but only 40 entries have been added to the stream file, the header will now reflect the $\texttt{totalLength}$ to be 1752. This means only $\texttt{40}$ of the $\texttt{100}$ entries of the *previously uncommitted operation $\mathtt{OpA}$ got overwritten, but the actual stream is still $\texttt{1812}$. How is this not a problem from an application point of view? It's because if a stream client requests for the stream, the stream server sends the stream only up to the $\texttt{totalLength}$ recorded in the header of the stream file, 1752, and not the actual length of the stream, $\texttt{1812}$. Concluding remarks The basic trick here is for the stream server to only use the information recorded in the header of the stream file, and to change that information only if an atomic operation is committed. This way the stream server always sends the stream only up to the last entry of the committed operation. All-in-all, this is just an optimal way to rollback. There's no need to delete information from the stream file. The header of the stream file is updated only if an atomic operation has been committed. This is the main reason why parameters such as the $\texttt{totalLength}$ and $\texttt{totalEntries}$ are recorded in the header of the stream file. # server-source-library.md: Interaction between the stream source and each stream server is enabled by the server-source library, which is a Go library with six main functions for modifying or adding entries to operations. Send data functions When each of these functions is called, a corresponding message is generated and sent to the stream server: 1. $\texttt{StartAtomicOp}(\ )$ starts an atomic operation. When called, a message that amounts to saying: "start an atomic operation," is generated and sent from the stream source to the stream server. 2. $\texttt{AddStreamEntry(u32 entryType, u8[] data)}$ adds an entry to the atomic operation and returns an $\texttt{u64 entryNumber}$. When called, a message equivalent to saying: "Add an entry of this type, with this data, to the current atomic operation," is generated and sent to the stream server. 3. $\texttt{AddStreamBookmark(u8[] bookmark)}$ adds an entry to the atomic operation and returns an $\texttt{u64}$ $\texttt{entryNumber}$. 4. $\texttt{CommitAtomicOp}(\ )$ commits an operation $\texttt{Op}$ so that its entries can be sent to stream clients. When called, a message which is tantamount to saying: "All entries associated with the current operation have been sent, the operation ends with the last sent entry," is generated and sent to the stream server. 5. $\texttt{RollbackAtomicOp}(\ )$ rolls back an atomic operation. 6. $\texttt{UpdateEntryData(u64 entryNumber, u32 entryType, u8[] newData)}$ updates an existing entry. This function only applies to entries for which the atomic operation has not been committed. Query data functions The stream source uses the following functions of the stream server-source library to get information from the stream server. - $\texttt{GetHeader()}$: The stream source uses this function to query the header of a particular entry. The function returns, $\texttt{struct HeaderEntry}$. - $\texttt{GetEntry(u64 entryNumber)}$: This function is used to get an entry that corresponds to a given entry number. It returns, $\texttt{struct FileEntry}$. - $\texttt{GetBookmark(u8[ ] bookmark)}$: The stream source uses this function to get a bookmark. The function returns, $\texttt{u64 entryNumber}$. - $\texttt{GetFirstEventAfterBookmark(u8[ ] bookmark)}$: This function is used to get the first entry after a given bookmark. It returns, $\texttt{struct FileEntry}$. !!! tip Find out more about the DATA STREAMER INTERFACE (API). It's possible to create, using the stream source-server library, a stream source that connects with a server, opens and commits operations. # stream-file.md: Next is an explanation of the stream file structure. The stream file is created in a binary format instead of a text file. It has a header page and one or more data pages. The header page is first and has a fixed size of 4096 bytes. The data pages follow immediately after the header page, and the size of each data page is 1 MB. Data pages contain entries. If an entry does not fit in the remaining page space, it gets stored in the next page. This means the unused space in the previous data page gets filled with some padding. !Figure Header page Let's zoom into how the $\texttt{Header}$â€‹ page looks like. The $\texttt{HeaderEntry}$ consists of the following data; $\texttt{magicNumbers}$, $\texttt{packetType}$, $\texttt{headerLength}$, $\texttt{streamType}$, $\texttt{TotalLength}$, and $\texttt{TotalEntries}$â€‹. 1. The $\texttt{HeaderEntry}$ starts with an array of 16 bytes, called $\texttt{magicNumbers}$â€‹. $$ \texttt{u8[16] magicNumbers} $$ The $\texttt{magicNumbers}$ identify the application to which the data in the stream file belongs. In the Polygon zkEVM case, the $\texttt{magicNumbers}$ is the ASCII-encoding of these sixteen (16) characters: $\texttt{polygonDATSTREAM}$. 2. After the $\texttt{magicNumbers}$ comes the $\texttt{packetType}$, which indicates whether the current page is a $\texttt{Header}$ page or a $\texttt{Data}$ page. $$ \texttt{u8 packetType = 1 // 1: Header entry} $$ The $\texttt{packetType}$ for the $\texttt{Header}$ entry is $\texttt{1}$, but it is $\texttt{2}$ for the $\texttt{Data}$ entry and $\texttt{0}$ for a padding. !Figure 3. Included in the $\texttt{Header}$ page is the $\texttt{streamType}$â€‹, which has the same meaning as seen in the Server-source protocol: It indicates the application, or in particular, the stream source node to which the stream server should connect. $$ \texttt{u64 streamType // 1: zkEVM Sequencer} $$ As mentioned in the above line of code, $\texttt{streamType = 1}$ means the stream source node is the zkEVM Sequencer. 4. The $\texttt{streamType}$ is then followed by the $\texttt{TotalLength}$â€‹ , which is the total number of bytes used in the stream file. $$ \texttt{u64 TotalLength // Total bytes used in the file} $$ 5. After the $\texttt{TotalLength}$ is the $\texttt{TotalEntries}$â€‹â€‹, which is the total number of entries used in the file. $$ \texttt{u64 TotalEntries // Total number of data entries} $$ Data pages A data page contains entries and some padding. Since this is a $\texttt{data}$ page, and not a $\texttt{Header}$ page, the entries are preceded by $\texttt{packetType = 2}$, while the padding is preceded by $\texttt{packetType = 0}$â€‹. $$ \begin{aligned} 1.\quad &\texttt{u8 packetType // 2:Data entry, 0:Padding} \\ 2.\quad &\texttt{u32 Length // Total length of data entry} \\ 3.\quad &\texttt{u32 entryType // 0xb0:Bookmark, 1:Event1, 2:Event2,... } \\ 4.\quad &\texttt{u64 entryNumber // Entry number (sequence from 0)} \\ 5.\quad &\texttt{u8[] data} \end{aligned} $$ The $\texttt{packetType}$ is followed by the $\texttt{Length}$ of the data entry, then the $\texttt{entryType}$ of the entry. That is, whether it is a bookmark or an event entry. A bookmark $\texttt{entryType}$ is indicated by $\texttt{0xb0}$, while each event's $\texttt{entryType}$ is its position among a sequence of events. That is, each i-th event is of $\mathtt{entryType = i}$. The next value after the $\texttt{entryType}$ is the entry number, denoted by $\texttt{entryNumber}$. The next values in a data page are $\texttt{data}$. After the last entry in a data page, is the $\texttt{packetType = 0}$ and some padding for any unused space. !Figure # implement-egp-strat.md: In this section we provide an elaborate discussion on how Polygon zkEVM network ensures transactions are executed with the best gas price for the user, while incurring minimal or no losses. You will learn how to sign transactions with the appropriate gas price ensuring: - There is minimal likelihood for your transactions to be reverted. - The sequencer prioritizes your transactions for execution. How to avert rejection of transactions The first step is to ensure that users sign transactions with sufficient gas price, and thus ensure the transactions are included in the L2's pool of transactions. Polygon zkEVM's strategy is to use pre-execution of transactions so as to estimate each transaction's possible gas costs. Suggested gas prices Due to fluctuations in the L1 gas price, the L2 network polls for L1 gas price every 5 seconds. The polled L1 gas prices are used to determine the appropriate L2 gas price to suggest to users. Since the L1 gas price is likely to change between when a user signs a transaction and when the transaction is pre-executed, the following parameters are put in place: - A 5-minute interval of several suggested gas prices, called $\texttt{MinAllowedPriceInterval}$. - During the $\texttt{MinAllowedPriceInterval}$, the user's transactions can be accepted for pre-execution, provided the $\texttt{SignedGasPrice}$ is higher than the lowest suggested gas price in the interval. - The lowest among the suggested gas prices is called $\texttt{L2MinGasPrice}$. !Figure: minimum allowed gas interval All transactions such that $\texttt{SignedGasPrice} > \texttt{L2MinGasPrice}$ are accepted for pre-execution. The following parameters can be configured in the Polygon zkEVM node: - $\texttt{DefaultMinGasPriceAllowed}$ which is the default minimum gas price to suggest. - $\texttt{MinAllowedGasPriceInterval}$, as explained above, is the interval within which to find the lowest suggested gas price and compare it with the user's gas price in the transaction. - $\texttt{PollMinAllowedGasPriceInterval}$ is the interval to poll L1 in order to find the suggested L2 minimum gas price. - $\texttt{IntervalToRefreshGasPrices}$ is the interval to refresh L2 gas prices. More specifically, these are configured in the $\texttt{[Pool]}$ section of the configuration file. !Figure: How to avoid incurring losses in L2 There are three measures put in place to help avoid incurring gas price-induced losses in the L2 network: - Pre-execution of transactions. - The breakeven gas price: $\texttt{BreakEvenGasPrice}$. - The L2's net profit. Transaction pre-execution You can use pre-execution to estimate the L2 resources each transaction will spend when processed. These resources are measured in terms of counters in the zkEVM's ROM, but are converted to gas units for better UX. This is the stage where transactions are either discarded or stored in the pool database, a pool of transactions waiting to be processed by the sequencer. The price of posting transaction data to L1 is charged to the zkEVM network at a full L1 price. Although computational costs in L2 may be accurately estimated, in cases where there is a reduction in such costs due to fewer L2 resources being spent, the user may be justified to sign a transaction with a very low gas price. But by signing such a low gas price, the user runs the risk of exhausting their wei reserves when transaction data is posted to L1. So then, if the use of $\texttt{L1GasPriceFactor = 0.04}$ is the only precautionary measure the L2 network takes in computing suggested gas prices, the L2 network will most likely incur losses. A $\texttt{suggestedFactor = 0.15}$ is therefore used to calculate each suggested gas price: $$ \texttt{suggestedL2GasPrice} = \texttt{L1GasPrice} \cdot \texttt{suggestedFactor} $$ The need for such a factor originates from the fact that the sequencer is obliged to process every transaction stored in the pool database, irrespective of its $\texttt{SignedGasPrice}$ and the prevailing L1 gas price. Breakeven gas price Calculating the breakeven gas price is another measure used in the Polygon zkEVM network to mitigate possible losses. As explained before, the computation is split in two; costs associated with data availability, and costs associated with the use of resources when transactions are processed. Costs associated with data availability The cost associated with data availability is computed as, $$ \texttt{DataCost} \cdot \texttt{L1GasPrice} $$ where $\texttt{DataCost}$ is the cost in gas for data stored in L1. The cost of data in Ethereum varies according to whether it involves zero bytes or non-zero bytes. In particular, non-zero bytes cost 16 gas units, while zero bytes cost 4 gas units. Also, recall that the computation of the cost for non-zero bytes must take into account constants that appear in transactions but are not included in the RLP, which includes: - The signature, which consists of 65 bytes. - The previously defined $\texttt{EffectivePercentageByte}$, which consists of a single byte. This results in a total of 66 constantly present bytes. Taking everything into consideration, $\texttt{DataCost}$ can be computed as: $$ \texttt{DataCost} = (\texttt{TxConstBytes} + \texttt{TxNonZeroBytes}) \cdot \texttt{NonZeroByteGasCost} \\ +\ \texttt{TxZeroBytes} \cdot \texttt{ZeroByteGasCost} $$ where $\texttt{TxNonZeroBytes}$ represents the count of non-zero bytes in a raw transaction, and similarly $\texttt{TxZeroBytes}$ represents the count of zero bytes in a raw transaction sent by the user. Computational costs Costs associated with transaction execution is denoted by $\texttt{ExecutionCost}$, and is measured in gas. In contrast to costs for data availability, calculating computational costs requires executing transactions. So then, $$ \texttt{GasUsed} = \texttt{DataCost} + \texttt{ExecutionCost} $$ The total fees received by L2 are calculated with the following formula: $$ \texttt{GasUsed} \cdot \texttt{L2GasPrice} $$ where $\texttt{L2GasPrice}$ is obtained by multiplying $\texttt{L1GasPrice}$ by a chosen factor less than 1, $$ \texttt{L2GasPrice} = \texttt{L1GasPrice} \cdot \texttt{L1GasPriceFactor} $$ In particular, we choose a factor of 0.04. Total price of a transaction The total transaction cost is simply the sum of data availability and computational costs: $$ \texttt{TotalTxPrice} = \big(\texttt{DataCost} \cdot \texttt{L1GasPrice} \big) + \big(\texttt{GasUsed} \cdot \texttt{L1GasPrice} \cdot \texttt{L1GasPriceFactor} \big) $$ In order to establish the gas price at which the total transaction cost is covered, we can compute $\texttt{BreakEvenGasPrice}$ as the following ratio: $$ \texttt{BreakEvenGasPrice} = \frac{\texttt{TotalTxPrice}}{\texttt{GasUsed}} $$ Additionally, we incorporate a factor $\texttt{NetProfit â‰¥ 1}$ that allows us to achieve a slight profit margin: $$ \texttt{BreakEvenGasPrice} = \frac{\texttt{TotalTxPrice}}{\texttt{GasUsed}} \cdot \texttt{NetProfit} $$ We then conclude that it is financially safe to accept the transaction if $$ \texttt{SignedGasPrice} > \texttt{BreakEvenGasPrice}. $$ However, a problem arises: In the RPC component, weâ€™re only pre-executing the transaction, meaning weâ€™re using an incorrect state root. Consequently, the $\texttt{GasUsed}$ is only an approximation. This implies that we need to multiply the result by a chosen factor before comparing it to the signed price to hedge against unforeseen costs. This ensures that the costs are covered in case more gas is ultimately required to execute the transaction. This factor is named $\texttt{BreakEvenFactor}$. Now we can conclude that if $$ \texttt{SignedGasPrice} > \texttt{BreakEvenGasPrice} \cdot \texttt{BreakEvenFactor} $$ then it is safe to accept the transaction. Observe that we still need to introduce gas price prioritization, which we explain later. Example (Breakeven gas price) Recall the example proposed before, where the $\texttt{GasPriceSuggested}$ provided by RPC was 2.85 gwei per gas, but the user ended up setting $\texttt{SignedGasPrice}$ to 3.3. The figure below depicts the current situation. !Figure: Timeline current L1GasPrice Suppose the user sends a transaction that has 200 non-zero bytes, including the constants and 100 zero bytes. Moreover, on pre-executing the transaction without an out-of-counters (OOC) error, $60,000$ gas units are consumed. Recall that, since we are using a "wrong" state root, this amount is only an estimation. Hence, using the previously explained formulas, the total transaction cost is: $$ \begin{aligned} &\texttt{TotalTxPrice} = \texttt{DataCost} \cdot \texttt{L1GasPrice} + \texttt{GasUsed} \cdot \texttt{L1GasPrice} \cdot \texttt{L1GasPriceFactor}\\ &\implies \texttt{TotalTxPrice} = (200 Â· 16 + 100 Â· 4) Â· 21 + 60, 000 Â· 21 Â· 0.04 = 126, 000\ \texttt{GWei} \end{aligned} $$ Observe that the 21 appearing in the substitution is the $\texttt{L1GasPrice}$ at the time of sending the transaction. Now, we are able to compute the $\texttt{BreakEvenGasPrice}$ as: $$ \texttt{BreakEvenGasPrice} = \frac{\texttt{TotalTxPrice}}{\texttt{GasUsed}} = \frac{126,000\ \texttt{GWei}}{60,000\ \texttt{Gas}} \cdot 1, 2 = 2.52\ \texttt{ GWei/Gas} $$ We have introduced a $\texttt{NetProfit}$ value of 1.2, indicating a target of a 20% gain in this process. At first glance, we might conclude the transaction has been accepted: $$ \texttt{SignedGasPrice} = 3.3 > 2.52 $$ but, recall that this is only an estimation, the gas consumed with the correct state root can differ. To avoid this, we introduce a $\texttt{BreakEvenFactor}$ of 30% to account for estimation uncertainties: $$ \texttt{SignedGasPrice} = 3.3 > 3.276 = 2.52 Â· 1.3 = \texttt{BreakEvenGasPrice} \cdot \texttt{BreakEvenFactor} $$ Consequently, the system accepts the transaction. Example (Breakeven factor) Suppose we disable the $\texttt{BreakEvenFactor}$ by setting it to 1. Our original transactionâ€™s pre-execution consumed $60,000$ gas: $$ \texttt{GasUsedRPC} = 60, 000 $$ However, let's assume the correct execution at the time of sequencing consumes $35,000$ gas. If we recompute $\texttt{BreakEvenGasPrice}$ using this updated used gas, we get $3.6\ \texttt{GWei/Gas}$, which is way higher than the original estimation. That means we should have charged the user a higher gas price in order to cover the whole transaction cost, standing at $105,000\ \texttt{GWei}$. But, since we are accepting all the transactions that sign more than 2.85 of gas price, we do not have any margin to increase it. In the worst case we are losing: $$ 105, 000 âˆ' 35, 000 Â· 2.85 = 5,250\ \texttt{GWei} $$ By introducing $\texttt{BreakEvenFactor}$, we are limiting the accepted transactions to the ones with, $$ \texttt{SignedGasPrice} â‰¥ 3.27 $$ in order to compensate for such losses. In this case, we have the flexibility to avoid losses and adjust both the user's and Polygon zkEVM network's benefits since: $$ 105, 000 âˆ' 35, 000 Â· 3.27 < 0 $$ Final Note: In the above example, even though we assume that a decrease in the $\texttt{BreakEvenGasPrice}$ is a result of executing with a correct state root, it can also decrease significantly due to a substantial reduction in $\texttt{L1GasPrice}$. # index.md: Here's how you can accurately determine the gas price to sign your transactions with when submitting transactions to the Polygon zkEVM network. Polygon zkEVM implements a mechanism called the Effective Gas Price (EGP) that guarantees fair gas fees in the interest of both the user and the network. Now that the EGP is in place, you can reduce any chance for a transaction revert, while making sure that accepted transactions receive your preferred prioritization. Meanwhile, the zkEVM network incurs minimal to no loss. How gas fees work in Ethereum In Ethereum, there are two adjustable parameters that have a direct impact on transaction gas fees: - The $\texttt{gasLimit}$, which is the maximum amount of gas units the user is willing to buy in order for their transactions to be included in a block and processed on chain. - The $\texttt{gasPrice}$, that is, the amount of ETH a user is willing to pay for 1 gas unit. For instance, a gas price of 10 Gwei means the user is willing to pay 0.00000001 ETH for each unit of gas. At the beginning of each transaction, the amount of wei sufficient to cover transaction costs is deducted from the user's account balance. That amount of wei is calculated as: $$ \texttt{gasLimit} \cdot \texttt{gasPrice} $$ The actual amount spent for the transaction is given by: $$ \texttt{gasUsed} \cdot \texttt{gasPrice} $$ And thus, any refund to the user is simply: $$ \texttt{gasLimit} \cdot \texttt{gasPrice} - \texttt{gasUsed} \cdot \texttt{gasPrice} $$ Transactions get reverted if the $\texttt{gasUsed}$ is greater than the $\texttt{gasLimit}$. Otherwise, the transaction is successful. Computing L2 gas fees The L2 gas price cannot simply be set to be the same as the L1 gas price (especially in the case of rollups where the goal is to reduce gas fees). Hence, we make the distinction between the two gas prices, and denote them as $\texttt{L2GasPrice}$ and $\texttt{L1GasPrice}$ respectively. It is important to calculate the appropriate L2 gas price while ensuring that transactions are successfully executed. Although the same formula is used, that is, $$ \texttt{gasLimit} \cdot \texttt{gasPrice} $$ and success is guaranteed if $\texttt{gasLimit}$ is greater than $\texttt{gasUsed}$, the gas used is determined by the gas cost for data availability plus the gas cost for transaction execution in L2. That is, $$ \texttt{gasUsed} = \texttt{DataCost} + (\texttt{L2 Execution gas cost}) $$ The total fees paid by the user is given by: $$ \texttt{TotalTxPrice} = \texttt{DataCost} \cdot \texttt{L1GasPrice} + (\texttt{L2 Execution gas cost}) \cdot \texttt{L2GasPrice} $$ Note that data availability is charged in L1 using the prevailing L1 gas price at the time of posting data. The main challenge is adjusting $\texttt{L2GasPrice}$ in terms of the $\texttt{L1GasPrice}$ to account for L2 resources spent when processing transactions. The general strategy is to use an $\texttt{L1GasPriceFactor}$ such that $$ \texttt{L2GasPrice} = \texttt{L1GasPrice} \cdot \texttt{L1GasPriceFactor} $$ Example (L1 gas price factor) For a transaction with an L1 gas price of 20 Gwei, the L2 gas price in the Polygon zkEVM network is calculated using a 4% factor as follows: $$ \texttt{L2GasPrice}\ =\ 20\ \text{Gwei}â‹…0.04=0.8\ \text{Gwei} $$ Current L2 fees can be viewed here https://l2fees.info. The $\texttt{L1GasPriceFactor}$ is used in the Polygon zkEVM network and is set to 0.04â€‹. There are a few complications that need to be carefully considered. There are 3 scenarios we aim to avoid when determining the $\texttt{L2GasPrice}$ to sign transactions with: - Transactions getting rejected due to the $\texttt{SignedGasPrice}$ being less than L2's minimum expected gas price ($\texttt{L2MinGasPrice}$). - Incurring losses in the L2 network because of high transaction gas costs. - Transactions receiving the least priority for sequencing. # tx-prioritization-egp.md: Users can sign transactions with gas prices higher than the suggested gas price for prioritized sequencing. That is, $$ \texttt{SignedGasPrice} > \texttt{suggestedGasPrice} $$ The priority ratio is defined as: $$ \texttt{PriorityRatio} = \frac{\texttt{SignedGasPrice}}{\texttt{suggestedGasPrice}} - 1 $$ Any transaction with $\mathtt{SignedGasPrice \leq GasPriceSuggested}$, implies that the user has chosen not to have the transaction prioritized. Computing effective gas price The computation of the effective gas price (or EGP) involves both the break even gas price and the priority ratio: $$ \texttt{EffectiveGasPrice} = \texttt{BreakEvenGasPrice} \cdot \big(1 + \texttt{PriorityRatio}\big) $$ Among the transactions stored in the pool database, the transactions with higher $\texttt{effectiveGasPrice}$ are prioritized at the time of sequencing due to the added $\texttt{PriorityRatio}$. Gas consumption deviations Since the actual gas consumed can deviate from the estimated gas consumption, we denote the estimated $\texttt{EGP}$ and the 'new' $\texttt{EGP}$ as $\texttt{EEGP}$ and $\texttt{NEGP}$, respectively. The extent of the deviation can be computed as a percentage: $$ \frac{|{\texttt{NEGP} - \texttt{EEGP}}|}{\texttt{EEGP}} \cdot 100 $$ The deviation percentage is compared to a parameter called $\texttt{FinalDeviationParameter}$, which is set to 10. This presents 2 scenarios and their corresponding consequences: 1. If the percentage deviation is lower than the final deviation parameter, $$ \frac{|\texttt{NEGP} âˆ' \texttt{EEGP}|}{\texttt{EEGP}} \cdot 100 < \texttt{FinalDeviationParameter} = 10, $$ it indicates that there is minimal distinction between charging the user with $\texttt{NEGP}$ compared to $\texttt{EEGP}$. Despite potential losses to the network, the user gets charged the $\texttt{EEGP}$ amount as the gas price. 2. On the contrary, if the percentage deviation equals or exceeds the deviation parameter, $$ \frac{|\texttt{NEGP} âˆ' \texttt{EEGP}|}{\texttt{EEGP}} \cdot 100 â‰¥ \texttt{FinalDeviationParameter} = 10, $$ the difference between executions can be so big it warrants adjustment of the gas price to be $\texttt{NEGP}$â€‹, and thus mitigate for potential losses to the network. Effective percentage The last parameter called the $\texttt{EffectivePercentage}$ is used to measure the unused portion of the user's signed gas price. In order to calculate the $\texttt{EffectivePercentage}$, one option is to consider pricing resources based on the number of consumed counters within the Polygon zkEVM's proving system. However, understanding this metric can be challenging for users because stating the efficiency through counters is a bit complicated. In favor of better UX, a formula involving gas is applied as it is more user-friendly. The primary objective is to compute $\texttt{EffectivePercentage}$ exclusively using gas, while allowing users to prioritize their transactions through the use of gas price, without the need for complex considerations such as used ROM counters. The effective percentage is computed as follows: $$ \texttt{EffectivePercentage} = \frac{ \texttt{GasPriceFinal}}{ \texttt{SignedGasPrice}} $$ where $\texttt{GasPriceFinal}$ is the gas price charged at the end of the entire processing by the sequencer. Note that the amount of wei that the user is charged for processing their transactions can be adjusted by* modifying $\texttt{GasPriceFinal}$. This $\texttt{EffectivePercentage}$ is provided by the sequencer as a single byte: $$ \texttt{EffectivePercentageByte} \in \{ 0, 1, . . . , 255 \} $$ which is computed from the $\texttt{EffectivePercentage}$ as follows: $$ \texttt{EffectivePercentageByte} = \lfloor \texttt{EffectivePercentage} Â· 256 \rfloor âˆ' 1 $$ Since having $\texttt{EffectivePercentage}$ implies having $\texttt{EffectivePercentageByte}$ and vice versa, the two terms are used interchangeably. So, $\texttt{EffectivePercentageByte}$ is often referred to as $\texttt{EffectivePercentage}$. Example (Effective percentage) Setting an $\texttt{EffectivePercentageByte}$ of $255\ (= \texttt{0xFF})$ means the $\texttt{EffectivePercentage} = 1$. In which case the user would pay the gas price they signed with, when sending the transaction, in total. That is: $$ \texttt{GasPriceFinal} = \texttt{SignedGasPrice} $$ In contrast, setting $\texttt{EffectivePercentageByte}$ to 127 means: $$ \texttt{EffectivePercentage} = 0.5 $$ Thus, only half of the gas price the user signed with gets charged as the transaction cost: $$ \texttt{GasPriceFinal} = \frac{\texttt{SignedGasPrice}}{2} $$ The transaction execution incurs only half of the signed gas price. Concluding remarks The effective gas price scheme, as outlined above, although steeped in details, takes all necessary factors and eventualities into consideration. Ultimately, the scheme is accurate and fair to both the users and the zkEVM network. Check out this repo for a detailed example of how the effective gas price is calculated. The content of this document series was sourced from the User fees document. # zkevm-egp-strat.md: This document presents an outline of Polygon zkEVM's strategy for executing transactions with the most accurate effective gas price. - Poll for L1 gas price regularly Since the L1 gas price fluctuates and the L2 gas price relies on it, it is necessary to query and fetch the L1 gas price frequently and thus have the most recent value at any given point in time. The Polygon zkEVM polls for the L1 gas price in regular intervals of 5 seconds. The polled gas prices are used to set the expected 'signed gas price', called $\texttt{MinL2GasPrice}$. - Provide suggested gas prices The polled L1 gas price values are sent to users upon request individually. A grace time interval of 5 minutes, called $\texttt{MinAllowedPriceInterval}$, is given to the user. It is recommended that the user sign their transactions with a gas price greater than the lowest of the gas prices fetched within the 5-minute interval. Otherwise, the transaction is rejected at the RPC pre-execution stage. - Pre-execute transactions at the RPC level Pre-execution of transactions involves: (a) Estimation of each transaction's possible consumption of L2 resources. That is, determining an approximate gas cost. (b) Checking userâ€™s signed gas price against the expected $\texttt{MinL2GasPrice}$. Store the transaction in the transaction pool manager if $\texttt{SignedGasPrice} < \texttt{MinL2GasPrice}$â€‹. Otherwise discard it. (c\) The transaction pool manager here refers to a collection of transactions waiting to be selected for execution by the sequencer. - Put in place a criterion for determining which transactions to store in the transaction pool manager Only the transactions that satisfy the criterion are stored on the transaction pool manager. The user's signed gas price is checked against either the breakeven factor, or the gas price suggested to the user. - Establish a criterion for when to execute transactions with user's signed gas price Some users' signed gas prices may be significantly higher than the effective gas price. In such cases, the sequencer can execute transactions with a much lower gas price to help save gas fees. Hence, there's a need for a criterion that determines whether a transaction gets executed with the user's signed gas price, or the effective gas price as per the RPC estimation. - Set a criterion for when to execute transactions with RPC-estimated EGP The effective gas price computed with the RPC-estimated gas price could be a lot higher than the actual gas price computed with the current state. In this case, the system can further optimize gas usage and help the user save on gas fees. The strategy here is to have a criterion that determines whether a transaction should get executed with the RPC-estimated effective gas price (EEGP) or the new effective gas price (NEGP), which results from the actual gas price. - Checking whether transactions include special opcodes The presence of opcodes such as $\texttt{GASPRICE}$ and $\texttt{BALANCE}$ in transactions can result in higher gas usage. zkEVM executes such transactions with the user's signed gas price. - Enhancing prioritization of transactions Since transactions are sequenced in decreasing order of the specified gas price, with higher preference given to large values, users need to provision sufficient gas price to allow for prioritization of transactions according to their needs. !Figure: Pre-excution scheme # index.md: Next, we provide a concise description of Polygon zkEVM's approach to calculating the effective gas price. It's an overview of the end-to-end flow of transactions, commencing from users signing transactions and submitting them at the RPC level, to when the sequencer executes them. It's a quicker way to go through all involved formulas or criteria, as well as examining explanatory examples. The end-to-end flow of transactions occurs in two phases: the RPC flow, and the sequencer flow. # rpc-flow-egp.md: The RPC flow phase of transactions consists of two stages: - The gas price suggestion. - Pre-execution of transactions. This flow ends with transactions being stored in a pool waiting to be executed by the sequencer. Gas price suggestion The L2 network (the zkEVM) polls for L1 gas price values and uses them to: - Suggest L2 gas price to users as per user requests. - Sets the minimum acceptable L2 gas price, denoted by $\texttt{L2MinGasPrice}$. The user then signs transactions with the appropriate gas price, called $\texttt{GasPriceSigned}$, based on the suggested L2 gas price, $\texttt{GasPriceSuggested}$. Transactions are accepted for pre-execution only if $$ \texttt{GasPriceSigned} > \texttt{L2MinGasPrice} $$ Pre-execution of transactions Pre-execution of transactions, which happens at the RPC level, involves estimating the gas required for processing the transactions submitted by the users. This is internally measured (internal to the zkEVM) in terms of resources spent to execute the transactions. These resources are the numbers of counters used up in the zkEVM ROM. A transaction is said to be out of counters (OOC) if the signed gas price is insufficient to pay for the required gas units. OOC transactions get rejected straight away, while those with no OOC stand a chance to be added to the pool. At this stage of the flow, the RPC also computes the "breakeven gas price", denoted by $\texttt{BreakEvenGasPriceRPC}$. That is, $$ \texttt{BreakEvenGasPrice} = \frac{\texttt{TotalTxPrice}}{\texttt{GasUsedRPC}} \cdot \texttt{NetProfit}, $$ where $\texttt{NetProfit}$ is the L2's marginal profit for transaction processing. Transactions with no OOC get added to the pool of transactions if, - Either $\texttt{GasPriceSigned} > \texttt{BreakEvenGasPriceRPC} \cdot \texttt{BreakEvenFactor}$. where $\texttt{GasUsedRPC}$ is the RPC's estimated gas cost. - Or $\texttt{GasPriceSigned} \geq \texttt{GasPriceSuggested}$. The total fees paid by the user is given by: $$ \texttt{TotalTxPrice} = \texttt{DataCost} \cdot \texttt{L1GasPrice} + (\texttt{L2 Execution gas cost}) \cdot \texttt{L2GasPrice} $$ The RPC flow is summarized in the figure below. !Figure: RPC flow Example (RPC tx flow) Consider a scenario where a user sends a query for a suggested gas price during a 5-minute interval, as shown in the figure below. Values of L1 gas prices, polled every 5 seconds, are displayed above the timeline, while the corresponding L2 gas prices are depicted below the timeline. See the figure below. !Figure: Suggested gas price (first) 1. Observe that, in the above timeline, the user sends a query at the time indicated by the dotted-arrow on the left. And that's when $\texttt{L1GasPrice}$ is 19. The RPC node responds with a $2.85 \texttt{ GWei/Gas}$, as the value of the suggested L2 gas price. This value is obtained as follows: $$ \texttt{GasPriceSuggested} = 0.15 \cdot 19 = 2.85 \texttt{ GWei/Gas} $$ where 0.15 is the zkEVM's suggested factor. 2. Let's suppose the user sends a transaction signed with a gas price of 3. That is, $\texttt{SignedGasPrice} = 3$. However, by the time the user sends the signed transaction, the L1 gas price is no longer 19 but 21. And its correponding suggested gas price is $\mathtt{3.15 = 21 \cdot 0.15}$. Note that the minimum suggested L2 gas price, in the 5-min time interval, is 2.85. And since $$ \texttt{SignedGasPrice} = 3 > 2.85 = \texttt{L2MinGasPrice} $$ the transaction gets accepted for pre-execution. 3. At this point, the RPC makes a request for pre-execution. That is, getting an estimation for the gas used, computed with a state root that differs from the one used when the transaction is sequenced. In this case, suppose an estimation of gas used is $\texttt{GasUsedRPC} = 60,000$, without an out of counters (OOC) error. 4. Since there's no out of counters (OOC) error, the next step is to compute the $\texttt{BreakEvenGasPriceRPC}$. Suppose it works out to be: $$ \texttt{BreakEvenGasPriceRPC} = 2.52\ \texttt{GWei/Gas} $$ (Details on how this calculation are covered later in the Implementing EGP strategy section.) 5. As noted in the outline of the RPC transaction flow, one more check needs to be done. That is, testing whether: $$ \texttt{SignedGasPrice} > \texttt{BreakEvenGasPriceRPC} \cdot \texttt{BreakEvenFactor} $$ Using the $\texttt{BreakEvenFactor} = 1.3$ yields: $$ \texttt{BreakEvenGasPrice} \cdot \texttt{BreakEvenFactor} = 2.52 \cdot 1.3 = 3.276 $$ And since $\texttt{SignedGasPrice} = 3 < 3.276$, the transaction is not immediately stored in the transaction pool. 6. However, since $$ \texttt{SignedGasPrice} = 3 â‰¥ 2.85 = \texttt{GasPriceSuggested} $$ and despite the risk of the network sponsoring the transaction, it is included in the transaction pool. # sequencer-flow-egp.md: In this phase of the end-to-end transaction flow, transactions go through different stages, depending on the user's $\texttt{GasPriceSigned}$: 1. Sequencing transactions coming from the transaction pool manager. 2. Estimating the effective gas price (EEGP) using the current $\texttt{L1GasPrice}$ and RPC's estimated $\texttt{GasUsedRPC}$. $$ \texttt{EffectiveGasPrice} = \texttt{BreakEvenGasPrice} \cdot \big(1 + \texttt{PriorityRatio}\big) $$ where the priority ratio is given by: $$ \texttt{PriorityRatio} = \frac{\texttt{SignedGasPrice}}{\texttt{suggestedGasPrice}} - 1 $$ Depending on whether $\texttt{GasPriceSigned} \leq \texttt{EEGP}$ or not, transactions get executed with either the $\texttt{GasPriceSigned}$ or the $\texttt{EEGP}$. 3. Amongst the transactions that are computed with the $\texttt{EEGP}$, further gas savings can be achieved by: - Computing the new effective gas price (NEGP), using the current state and the EEGP. - Calculating the gas consumption deviation percentage and comparing it to a fixed deviation parameter. i.e., $\texttt{FinalDeviationParameter} = 10$. - There's no further execution if the gas consumption deviation percentage is less than the fixed parameter, $\texttt{FinalDeviationParameter} = 10$. - Otherwise, check if $\texttt{GasPriceSigned} \leq \texttt{NEGP}$. If true, then execute transactions again using the $\texttt{GasPriceSigned}$. If false, continue to the next stage. 4. Checking the usage of $\texttt{GASPRICE}$ and $\texttt{BALANCE}$ opcodes. - Transactions with these two opcodes get executed with the $\texttt{GasPriceSigned}$. - Otherwise, they are executed with the $\texttt{NEGP}$. Since the sequencer is obliged to execute all transactions in the transaction pool manager, each transaction is executed during a particular stage of the flow described above. The entire sequencer flow is summarized in the figure below. !Figure: Sequencer flow Example (Sequencer flow) Let's continue the numerical example we have been using throughout this document. As seen in previous examples, the figure below displays L1 gas prices above the timeline, while the associated suggested L2 gas prices are shown below the timeline. At the time of sequencing the transaction, the suggested gas price is given by, $$ \texttt{GasPriceSuggested} = 0.15 \cdot \texttt{L1GasPrice} $$!Figure: 1. Suppose the user signed a gas price of $3.3\ \texttt{GWei/Gas}$. Recall how we previously obtained the $\texttt{BreakEvenGasPriceRPC}$ of $2.52\ \texttt{GWei/Gas}$. According to the figure above, the network recommends a gas price of 3, which corresponds to an L1 gas price of 20. This results in the following priority factor: $$ \begin{aligned} \texttt{PriorityRatio} &= \frac{\texttt{SignedGasPrice}}{\texttt{SuggestedGasPrice}} - 1 \\ \text{ } &= \frac{3.3}{3} - 1 = 0.1 \end{aligned} $$ and an estimated effective gas price: $$ \begin{aligned} \texttt{EEGP} &= \texttt{BreakEvenGasPrice} \cdot \big(1 + \texttt{PriorityRatio}\big) \\ \text{ } &= 2.52 \cdot \big(1 + 0.1 \big) = 2.722\ \texttt{GWei/Gas} \end{aligned} $$ This amounts to a 10% increment to the gas price for transaction prioritization. 2. Since the signed gas price is bigger than the estimated effective gas price, $$ \texttt{SignedGasPrice} = 3.3 > 2.772 = \texttt{EEGP} $$ the transaction can be executed with the $\texttt{SignedGasPrice}$. 3. The sequencer can use the current and correct state, together with the computed $\texttt{EEGP}$, in order to obtain a more accurate measure of the gas used, call it $\texttt{GasUsedNew}$. Suppose that, in this case, we obtain $$ \texttt{GasUsedNew} = 95,000\ \texttt{Gas} $$ which is bigger than the RPC-estimated gas of $60,000$. 4. With the new $\texttt{GasUsedNew}$, an adjusted effective gas price ($\texttt{NEGP}$) can be computed by the following steps. Firstly, the total transaction cost: $$ \texttt{TxCostNew} = (200 Â· 16 + 100 Â· 4) Â· 20 + 95,000 Â· 20 Â· 0.04 = 148,000 \ \texttt{GWei} $$ We assume that the transaction has 200 non-zero bytes and 100 zero bytes. Secondly, the new breakeven gas price: $$ \texttt{BreakEvenGasPriceNew} = \frac{148, 000}{95, 000} Â· 1.2 = 1.869\ \texttt{GWei/Gas} $$ where a 20% breakeven factor is applied. Thirdly, the new effective gas price: $$ \begin{aligned} \texttt{NEGP} &= \texttt{BreakEvenGasPriceNew} \cdot \big(1 + \texttt{PriorityRatio}\big) \\ \text{ } &= 1.869 Â· \big(1 + \big(\frac{\texttt{SignedGasPrice}}{\texttt{SuggestedGasPrice}} - 1 \big) \big) \\ \text{ } &= 1.869 Â· \big(1 + \big(\frac{3.3}{3} - 1 \big) \big) \\ \text{ } &= 1.869 Â· 1.1 = 2.056\ \texttt{GWei/Gas} \end{aligned} $$ Observe that the transaction cost is much higher than the RPC-estimation of $126,000$, even when the L1 gas price has decreased from 21 to 20 due to a huge increase in gas. 5. Observe that there is a significant deviation between both effective gas prices: $$ \frac{|\texttt{NEGP} âˆ' \texttt{EEGP}|}{\texttt{EEGP}} \cdot 100 = 25.82 > 10 $$ Executing the transaction with the signed gas price, while the deviation is this big, amounts to penalizing the user unfairly. Thus, since the new EGP is smaller than the signed gas price: $$ \texttt{SignedGasPrice}\ = 3.3 > 2.52 =\ \texttt{EEGP} â‰« 2.056 =\ \texttt{NEGP} $$ instead of charging the signed gas price, a further attempt to adjust the gas price is made by charging the $\texttt{NEGP} = 2.056$ to the user. 6. In the case where the transaction has none of the two opcodes, $\texttt{GASPRICE}$ and $\texttt{BALANCE}$, in the source address opcodes, the transaction gets executed with the NEGP: $$ \texttt{GasPriceFinal} = \texttt{NEGP} = 2.056\ \texttt{GWei/Gas} $$ Observe that $\texttt{GasUsedFinal}$ should be the same as $\texttt{GasUsedNew} = 95,000$. Finally, the $\texttt{EffectivePercentage}$ and $\texttt{EffectivePercentageByte}$ are computed as follows: $$ \texttt{EffectivePercentage} = \frac{\texttt{GasPriceFinal}}{\texttt{SignedGasPrice}} = \frac{2.056}{3.3} = 0.623. $$ $$ \texttt{EffectivePercentageByte} = \lfloor \texttt{EffectivePercentage} Â· 256 \rfloor âˆ' 1 = 158 $$ Observe that the user has been charged 62.3% of the gas price they signed at the time of sending the transaction. # addresses.md: Mainnet contracts | Contract name | Address | | ---------------------------- | --- | | PolygonRollupManager | 0x5132A183E9F3CB7C848b0AAC5Ae0c4f0491B7aB2 | | PolygonZkEVMBridgeV2 | 0x2a3DD3EB832aF982ec71669E178424b10Dca2EDe | | PolygonZkEVMGlobalExitRootV2 | 0x580bda1e7A0CFAe92Fa7F6c20A3794F169CE3CFb | | FflonkVerifier | 0x4F9A0e7FD2Bf6067db6994CF12E4495Df938E6e9 | | PolygonZkEVMDeployer | 0xCB19eDdE626906eB1EE52357a27F62dd519608C2 | | PolygonZkEVMTimelock | 0xEf1462451C30Ea7aD8555386226059Fe837CA4EF | zkEVM contracts | Contract name | Address | | -------------------- | -- | | PolygonZkEVMBridgeV2 | 0x2a3DD3EB832aF982ec71669E178424b10Dca2EDe | | PolygonZkEVMTimelock | 0xBBa0935Fa93Eb23de7990b47F0D96a8f75766d13 | CDK contracts !!! warning TODO: --> Test private keys | Private key | Address | | -------------------- | -- | | 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80 | 0xf39fd6e51aad88f6f4ce6ab8827279cfffb92266 | | 0xdfd01798f92667dbf91df722434e8fbe96af0211d4d1b82bbbbc8f1def7a814f | 0xc949254d682d8c9ad5682521675b8f43b102aec4| # exit-roots.md: An exit tree is a binary, append-only, sparse Merkle tree (SMT) with a maximum depth of 32, where leaf nodes store bridging data. The Merkle root of an exit tree is known as the exit tree root, and it is the fingerprint of all the information recorded in the exit tree's leaf nodes. The global exit tree root of the L1 info tree is, therefore, the source of truth for the whole network. Rollup local exit trees The L2 bridge contract manages a special Merkle tree called a local exit tree for each network that participates in bridging and claiming which is updated by the PolygonZkEVMGlobalExitRootL2.sol contract.

!Local exit tree for network participant

Data from bridgeAsset() and bridgeMessage() calls on the bridge is stored in leaf nodes on the local exit trees. !!! important The following exit tree structures are managed by: - The PolygonRollupManager.sol. - The L1 PolygonZkEVMBridgeV2.sol contract. - ThePolygonZkEVMGlobalExitRootV2.sol. Exit tree for rollups The roots of the L2 local exit trees feed into a single exit tree that manages state from all participating L2 rollups. The state lives in the L1 realm and is accessed at sequencing time.

!Exit tree for rollups

The L2 local exit root is accessible on the rollup manager by calling the getRollupExitRoot() method. L1 local exit tree Every time there is a call to bridgeAsset() and bridgeMessage() on the bridge at the L1 Ethereum level, the data is stored in a leaf node on the L1 local exit tree.

!L1 local exit tree

L1 info tree The L1 info tree is stored in the PolygonZkEVMGlobalExitRootV2.sol contract also known as the global exit root manager. All subtrees exit roots feed into the leaves of the L1 info tree, which contains the global exit root (GER). The GER is the fingerprint of the information stored in all trees, and thus represents the global state of the system.

!Exit tree for rollups

Exit leaves Two constants define transaction leaf types in the bridge contract. solidity // Leaf type asset uint8 private constant LEAFTYPEASSET = 0; // Leaf type message uint8 private constant LEAFTYPEMESSAGE = 1; Data in a leaf contains a Keccak256 hash of the metadata (ABI encoded metadata if any) and the following parameters (matched by publicly available transaction data as seen in the bridge L1 to L2 documentation): solidity addLeaf(getLeafValue(LEAFTYPEASSET, // or LEAFTYPEMESSAGE originNetwork, originTokenAddress, destinationNetwork, destinationAddress, leafAmount, keccak256(metadata))); !!! info "Leaf parameters" - int32 originNetwork: Origin network ID, where the original asset belongs. - address originTokenAddress: If leafType = 0, Origin network token address (0x0000...0000) is reserved for ether. If leafType = 1, msg.sender of the message. - uint32 destinationNetwork: Bridging destination network ID. - address destinationAddress: Address that receives the bridged asset in the destination network. - uint256 leafAmount: Amount of tokens/ether to bridge. - bytes32 keccak256(metadata): Hash of the metadata. This metadata contains information about assets transferred or the message payload. Updating system state The system uses a set of exit tree roots to manage system state. Leaves of the trees point to transaction data such as detailed above. Adding a new leaf to the tree triggers an update to the exit tree root which then propagates to an update on the global exit tree root. Using Merkle tree exit roots in this way, referenced by the bridge contracts and accessible to the PolygonRollupManager contract with getters, the bridge contract triggers data synchronization across L1 and L2, including at the sequencer and state db level. The use of two distinct global exit root manager contracts for L1 and L2, as well as separate logic for the sequencing flow and the bridge contract, allows for extensive network interoperability. Meanwhile, all asset transfers can be validated by any L1 and L2 node due to the accessibility of state data. The exit roots are modified in two key flows; sequencing and bridging. Sequencing flow The PolygonZkEVMGlobalExitRootV2 contract manages updates to the exit roots on sequencing. The contract calls updateExitRoot(...) on the GlobalExitRootManager during the sequencing flow to add an exit leaf to the relevant exit tree. !Update exit roots via sequencing flow 1. Initiate update: PolygonZkEVMEtrog initiates the update process by calling updateExitRoots on PolygonRollupBaseEtrog. 2. Retrieve current roots: PolygonRollupBaseEtrog retrieves the current local and global exit roots from PolygonZkEVMGlobalExitRootL2 and PolygonZkEVMGlobalExitRootV2 respectively. 3. Compute new exit root: PolygonRollupBaseEtrog computes the new exit root based on the retrieved local and global exit roots. 4. Update local exit root: PolygonRollupBaseEtrog updates the local exit root in PolygonZkEVMGlobalExitRootL2. 5. Update global exit root: PolygonRollupBaseEtrog updates the global exit root in PolygonZkEVMGlobalExitRootV2. 6. Verify updated exit root: PolygonRollupBaseEtrog calls getRollupExitRoot on PolygonRollupManager to verify the updated exit root. !!! tip "L1 or L2 update" - If msg.sender is the bridge contract, the L1 local exit root is updated. - If msg.sender is the rollup manager, the L2 local exit root is updated. Bridging flow When bridging, the global exit root is updated if the forceUpdateGlobalExitRoot variable is set to true. !Update exit roots via bridging flow 1. The user interacts with the PolygonZkEVMBridgeV2 contract by calling the bridge() function. 2. PolygonZkEVMBridgeV2 calls updateLocalExitRoot() on PolygonZkEVMGlobalExitRootL2, which updates the local exit root. 3. If forceUpdateGlobalExitRoot is set to true, PolygonZkEVMBridgeV2 calls updateGlobalExitRoot() on PolygonZkEVMGlobalExitRootV2, which updates the global exit root. # index.md: Polygon zkEVM deploys smart contracts to manage transaction processing and related data transfers between itself and other networks, such as the Ethereum L1 or other L2s connected to the AggLayer. There are four key contract types built into the system design: - Consensus contracts. - Rollup manager. - Bridge contract. - Exit root management.

Consensus Consensus contracts live in the L1 and expose specific functions for controlling the sequencing and verification mechanisms triggered by the sequencer and aggregator. Polygon zkEVM can be implemented as, - Either a zk-rollup, deploying the PolygonZkEVMEtrog.sol consensus contract, where all transaction data is posted to Ethereum. - Or, a validium using a data-availability committee (DAC) for managing availability of transaction data, and deploying the PolygonValidiumEtrog.sol consensus contract. These contracts therefore define the type of a CDK chain, either a validium or zk-rollup. Rollup manager The PolygonRollupManager.sol contract is useful for managing CDK chains. It is responsible for creating, updating, and verifying CDK rollup and validium chains. Bridge The unified bridge contract PolygonZkEVMBridgeV2.sol is responsible for bridging and claiming assets or messages across L1 and L2 chains. Global exit roots Each bridging of an asset or a message from the zkEVM is recorded on a leaf of the zkEVM's local exit tree. Every update of the local exit tree at the leaf-level triggers an update of the local exit root, which in turn is appended to a leaf of the global exit tree. Similarly, each bridging of an asset or a message from the L1 to the zkEVM is recorded on a leaf of the L1 Info tree. The PolygonZkEVMGlobalExitRootV2.sol contract manages the exit roots across multiple networks at the Ethereum L1 level. The PolygonZkEVMGlobalExitRootL2.sol contract manages the zkEVM state by keeping track of the current local exit root. Validium stacks CDK validium stacks use the cdk-validium-contracts which has slightly adjusted behavior to take account of data-availability components and custom CDK requirements. The CDK repo is a fork of the zkEVM main contracts repo and all contracts, therefore, extend from common interfaces. !!! important - A CDK validium stack starts off as a rollup stack. - It may interchangeably be referred to as such when discussing aspects shared by the two options. # main-contracts.md: Consensus contracts The following contracts manage consensus mechanisms. They deal with sequencing and verifying transaction batches across the L1 and L2 networks. PolygonRollupBaseEtrog.sol PolygonRollupBaseEtrog.sol is the base contract for rollups and validiums. PolygonZkEVMEtrog.sol PolygonZkEVMEtrog.sol is the rollup contract inherited from the base contract. This contract calls the onSequenceBatches(...) function on the PolygonRollupManager.sol contract to trigger the verification mechanism after successful sequencing through the sequenceBatches(...) call.

!Polygon Solidity smart contract consensus contract set

PolygonValidiumEtrog.sol PolygonValidiumEtrog.sol is the validium contract inherited from the base contract. This contract calls the onSequenceBatches(...) function on the PolygonRollupManager.sol contract to trigger the verification mechanism after successful sequencing through the sequenceBatchesValidium(...) call. !!! info - Custom chain contracts also exist at the consensus level and extend from common consensus base contracts. Rollup manager PolygonRollupManager.sol The PolygonRollupManager.sol contract is responsible for managing rollups. It verifies batches, and also creates and updates rollup stacks by storing hash-sequenced data as newly sequenced batches arrive. It is responsible for the verification workflow by supplying updated exit root data to the PolygonZkEVMGlobalExitRootV2.sol contract.

!Polygon Solidity smart contract rollup manager{ width=80% }

Key functionality - Defining and adding rollup types which contains consensus implementation details and compatibility checks. - Defining the RollupData struct. - Initializes the trusted aggregator process for verifying multiple batches. - Getting exit root data by computing all local exit roots of all rollups. - Calculating batch rewards. Bridge PolygonZkEVMBridgeV2.sol The PolygonZkEVMBridgeV2.sol is the main communication mechanism between the L1 and L2 realms. It manages bridging and claiming of assets and messages across environments.

!Polygon Solidity smart contract bridge{ width=80% }

Key functionality - Bridging assets with the bridgeAsset(...) function. - Bridging messages with a choice of bridgeMessage(...) functions for various scenarios. - Claiming assets with the claimAsset(..) function. - Claiming messages with the claimMessage(...) function. - Verifying state and updating the global exit root with state changes. - Providing access to the global exit root manager via the IBasePolygonZkEVMGlobalExitRoot. - Interacting with the PolygonZkEVMGlobalExitRootL2.sol contract which exists in the L2 space as part of the bridge functionality. Exit roots PolygonZkEVMGlobalExitRootV2.sol The PolygonZkEVMGlobalExitRootV2.sol contract manages the L1 info tree that represents the current state of the system by updating global exit roots on state changes. It does this task across multiple networks and layers. Key functionality - Updating the L1 info tree by emitting the UpdateL1InfoTree(...) event. - Updating exit roots. - Retrieving latest exit roots and leaf values.

!Polygon Solidity smart contract exit root L1{ width=60% }

PolygonZkEVMGlobalExitRootL2.sol The PolygonZkEVMGlobalExitRootL2.sol contract manages the L2 rollup info trees. It is a lighter-weight version of the global exit root contract mentioned previously. Key functionality - Stores every global exit root in the globalExitRootMap where keys are global exit roots and values are timestamps. - Updates the lastRollupExitRoot on any bridge call. - Updates the L2 network and exit root with the updateExitRoot(...) function.

!Polygon Solidity smart contract exit root L2{ width=60% }

 # sequencing.md: Transactions flowing through the system reach the smart contract environment after one of two contract call use cases: - Sequence batches requests coming from the sequencer component in the node. - Verifying batches requests coming from the aggregator component in the node. This section focuses on the sequencing workflow. The diagram below shows the sequencing workflow for rollup (non-validium) stacks which calls sequenceBatches(...) and onSequenceBatches(...). !Polygon Solidity smart contract architecture sequenceBatches() The sequenceBatches() function is called on the PolygonZkEVMEtrog.sol contract: sequenceBatches(batches, maxSequenceTs, initSequenceBatch, l2Coinbase) The rollup sequencer component calls the sequenceBatches function on the PolygonZkEVMEtrog.sol contract which inherits the function from PolygonRollupBaseEtrog.sol. The function takes an array of BatchData structs from one of the consensus contracts. Each struct contains L2 Ethereum transactions data, and some forced state information. solidity struct BatchData { bytes transactions; bytes32 forcedGlobalExitRoot; uint64 forcedTimestamp; bytes32 forcedBlockHashL1; } The function validates arguments, checks and organizes the batches, and appends them in the correct sequence while computing an accumulated hash. Finally, the function emits a SequenceBatches event which sends a newly sequenced batch of transactions to the PolygonRollupManager.sol contract after the onSequenceBatches(...) function returns successfully. Stepwise, the function does the following: 1. Validates arguments. 1. Tells the bridge to update the global exit root by calling globalExitRootManager.updateExitRoot(L1LocalExitRoot) which creates a new global exit root with the newest L1 local exit root. 1. Gets L1 info root and other variables needed for computation. 1. Goes through the batches to compute the accumulated hash with keccak(batch.transaction) and keccak(accInputHash, txHash, l1InfoRoot, maxSequenceTs, l2Coinbase, bytes32(0)). 1. Stores the accumulated hash. 1. Caller pays the rollup manager in POL. 1. Calls the PolygonRollupManager.onSequenceBatches(...) function which waits for an OnSequenceBatches(...) event callback. 1. Emits SequenceBatches(...) event. onSequenceBatches() The onSequenceBatches() function is called on the PolygonRollupManager.sol contract: onSequenceBatches(newSequencedBatches, newAccInputHash) It takes the sequenced batches and the accumulated hash from the calling contract, adds the batches to the correct stack, and updates the batch count. Stepwise, the function does the following: 1. Validates the arguments and the caller contract. 1. Updates the totalSequencedBatches storage variable. 1. Updates the lastBatchSequenced and adds a new SequencedBatchData struct for the rollup that called sequenceBatches. 1. Attempts to consolidate pending state for the rollup by updating lastVerifiedBatch, batchNumToStateRoot[], and lastLocalExitRoot state variables, and also by updating globalExitRootManager.updateExitRoot(L2sLocalExitRoot), after which it emits a ConsolidatePendingState(...) event. 1. Emits an OnSequenceBatches(...) event back to the original sequenceBatches(...) call. sequenceBatchesValidium() The sequenceBatchesValidium() function is called on the PolygonValidiumEtrog.sol contract: sequenceBatchesValidium(batches, l2Coinbase, dataAvailabilityMessage) !!! info - This function is not included in the sequence diagram above. - The differences, however, are minimal. The sequencing logic is nearly the same as for the rollup sequenceBatches(...) function except the function takes a ValidiumBatchData[] array instead of BatchData[]. This means that, instead of passing the actual transaction data, the struct passes the hashes of the transactions. solidity struct ValidiumBatchData { bytes32 transactionsHash; bytes32 forcedGlobalExitRoot; uint64 forcedTimestamp; bytes32 forcedBlockHashL1; } It also has a dataAvailabilityMessage parameter instead of the sequence information. This parameter contains all the ECDSA address signatures of the committee in ascending order and is used for verification. Stepwise, the function has identical logic to the PolygonRollupBaseEtrog.sequenceBatches(...) function apart from the following steps: 1. ValdiumBatchData instead of BatchData. 1. Accumulates the txHash into accumulatedNonForcedTransactionHash. 1. Adds a validity check with dataAvailabilityProtocol.verifyMessage(accumulatedNonForcedTransactionHash, dataAvailabilityMessage). # verification.md: Transactions flowing through the system reach the smart contract environment after one of two contract call use-cases: - Sequence batches requests coming from the sequencer component in the node. - Verifying batches requests coming from the aggregator component in the node. This section focuses on the verification workflow. The sequence diagram below shows the verification workflow for rollup stacks and/or the AggLayer calling the verifyBatchesTrustedAggregator(...) function on the rollup manager. !Polygon Solidity consensus verification flow verifyBatchesTrustedAggregator() The verifyBatchesTrustedAggregator() function is called on the PolygonRollupManager contract: verifyBatchesTrustedAggregator(rollupID, pendingStateNum, initNumBatch, finalNewBatch, newLocalExitRoot, newStateRoot, beneficiary, proof) The zkEVM node aggregator, or the AggLayer, calls the verifyBatchesTrustedAggregator function on the PolygonRollupManager.sol contract. The function creates a rollup data storage object with the data provided by the caller, which it first verifies by sending it to the helper function verifyAndRewardBatches. This internal function calculates the inputSnark bytes value, which is a (SHA256 % "RFIELD") calculation on the input data, and uses this value to verify the proof by calling rollup.verifier.verifyProof(proof, [inputSnark]) on an IVerifierRollup interface implementation. !!! note Using a verifier variable on the rollup object means the verifier implementation can be customized in the future. Next, in the verifyAndRewardBatches helper function, the code reverts if the proof does not verify successfully. Otherwise, the code pays POL rewards to the beneficiary. The function then updates the state and calls rollup.rollupContract.onVerifyBatches(...), which is a callback into the rollup consensus contract, allowing for customizable behavior. onVerifyBatches(...) takes the new state root and final batch, and emits a VerifyBatches(...) event. The command flow returns to the verifyBatchesTrustedAggregator function which consolidates and updates the state with the newStateRoot and the new localExitRoot and calls the updateExitRoot(...) function on the GlobalExitRootManager contract passing the rollup's updated exit root. # PolygonRollupManager.md: Functions constructor solidity function constructor(contract IPolygonZkEVMGlobalExitRootV2 globalExitRootManager, contract IERC20Upgradeable pol, contract IPolygonZkEVMBridge bridgeAddress) public Parameters | Name | Type | Description | | :--- | :--- | :--- | |globalExitRootManager | contract IPolygonZkEVMGlobalExitRootV2 | Global exit root manager address |pol | contract IERC20Upgradeable | POL token address |bridgeAddress | contract IPolygonZkEVMBridge | Bridge address initialize solidity function initialize(address trustedAggregator, uint64 pendingStateTimeout, uint64 trustedAggregatorTimeout, address admin, address timelock, address emergencyCouncil, contract PolygonZkEVMExistentEtrog polygonZkEVM, contract IVerifierRollup zkEVMVerifier, uint64 zkEVMForkID, uint64 zkEVMChainID) external Parameters | Name | Type | Description | | :--- | :--- | :-- | |trustedAggregator | address | Trusted aggregator address |pendingStateTimeout | uint64 | Pending state timeout |trustedAggregatorTimeout | uint64 | Trusted aggregator timeout |admin | address | Admin of the rollup manager |timelock | address | Timelock address |emergencyCouncil | address | Emergency council address |polygonZkEVM | contract PolygonZkEVMExistentEtrog | New deployed Polygon zkEVM which will be initialized wiht previous values |zkEVMVerifier | contract IVerifierRollup | Verifier of the new zkEVM deployed |zkEVMForkID | uint64 | Fork id of the new zkEVM deployed |zkEVMChainID | uint64 | Chain id of the new zkEVM deployed addNewRollupType solidity function addNewRollupType(address consensusImplementation, contract IVerifierRollup verifier, uint64 forkID, uint8 genesis, bytes32 description) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |consensusImplementation | address | Consensus implementation |verifier | contract IVerifierRollup | Verifier address |forkID | uint64 | ForkID of the verifier |genesis | uint8 | Genesis block of the rollup |description | bytes32 | Description of the rollup type obsoleteRollupType solidity function obsoleteRollupType(uint32 rollupTypeID) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollupTypeID | uint32 | Rollup type to obsolete createNewRollup solidity function createNewRollup(uint32 rollupTypeID, uint64 chainID, address admin, address sequencer, address gasTokenAddress, string sequencerURL, string networkName) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollupTypeID | uint32 | Rollup type to deploy |chainID | uint64 | ChainID of the rollup, must be a new one |admin | address | Admin of the new created rollup |sequencer | address | Sequencer of the new created rollup |gasTokenAddress | address | Indicates the token address that will be used to pay gas fees in the new rollup Note if a wrapped token of the bridge is used, the original network and address of this wrapped will be used instead |sequencerURL | string | Sequencer URL of the new created rollup |networkName | string | Network name of the new created rollup addExistingRollup Add an already deployed rollup. solidity function addExistingRollup(contract IPolygonRollupBase rollupAddress, contract IVerifierRollup verifier, uint64 forkID, uint64 chainID, bytes32 genesis, uint8 rollupCompatibilityID) external !!! note - This rollup does not follow any rollupType. Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollupAddress | contract IPolygonRollupBase | Rollup address |verifier | contract IVerifierRollup | Verifier address, must be added before |forkID | uint64 | Fork id of the added rollup |chainID | uint64 | Chain id of the added rollup |genesis | bytes32 | Genesis block for this rollup |rollupCompatibilityID | uint8 | Compatibility ID for the added rollup addExistingRollup Add an already deployed rollup. solidity function addExistingRollup(contract IPolygonRollupBase rollupAddress, contract IVerifierRollup verifier, uint64 forkID, uint64 chainID, uint8 rollupCompatibilityID, uint64 lastVerifiedBatch) internal returns (struct PolygonRollupManager.RollupData rollup) !!! note - This rollup does not follow any rollupType. Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollupAddress | contract IPolygonRollupBase | Rollup address |verifier | contract IVerifierRollup | Verifier address, must be added before |forkID | uint64 | Fork id of the added rollup |chainID | uint64 | Chain id of the added rollup |rollupCompatibilityID | uint8 | Compatibility ID for the added rollup |lastVerifiedBatch | uint64 | Last verified batch before adding the rollup updateRollup Upgrade an existing rollup. solidity function updateRollup(contract ITransparentUpgradeableProxy rollupContract, uint32 newRollupTypeID, bytes upgradeData) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollupContract | contract ITransparentUpgradeableProxy | Rollup consensus proxy address |newRollupTypeID | uint32 | New rolluptypeID to upgrade to |upgradeData | bytes | Upgrade data onSequenceBatches Sequence batches, callback called by one of the consensus rollups managed by this contract. solidity function onSequenceBatches(uint64 newSequencedBatches, bytes32 newAccInputHash) external returns (uint64) Parameters | Name | Type | Description | | :--- | :--- | :--- | |newSequencedBatches | uint64 | Number of batches sequenced |newAccInputHash | bytes32 | New accumulate input hash verifyBatches Allows an aggregator to verify multiple batches. solidity function verifyBatches(uint32 rollupID, uint64 pendingStateNum, uint64 initNumBatch, uint64 finalNewBatch, bytes32 newLocalExitRoot, bytes32 newStateRoot, address beneficiary, bytes32[24] proof) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollupID | uint32 | Rollup identifier |pendingStateNum | uint64 | Init pending state, 0 if consolidated state is used |initNumBatch | uint64 | Batch which the aggregator starts the verification |finalNewBatch | uint64 | Last batch aggregator intends to verify |newLocalExitRoot | bytes32 | New local exit root once the batch is processed |newStateRoot | bytes32 | New State root once the batch is processed |beneficiary | address | Address that will receive the verification reward |proof | bytes32[24] | Fflonk proof verifyBatchesTrustedAggregator Allows a trusted aggregator to verify multiple batches. solidity function verifyBatchesTrustedAggregator(uint32 rollupID, uint64 pendingStateNum, uint64 initNumBatch, uint64 finalNewBatch, bytes32 newLocalExitRoot, bytes32 newStateRoot, address beneficiary, bytes32[24] proof) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollupID | uint32 | Rollup identifier |pendingStateNum | uint64 | Init pending state, 0 if consolidated state is used |initNumBatch | uint64 | Batch which the aggregator starts the verification |finalNewBatch | uint64 | Last batch aggregator intends to verify |newLocalExitRoot | bytes32 | New local exit root once the batch is processed |newStateRoot | bytes32 | New State root once the batch is processed |beneficiary | address | Address that will receive the verification reward |proof | bytes32[24] | Fflonk proof verifyAndRewardBatches Verify and reward batches internal function. solidity function verifyAndRewardBatches(struct PolygonRollupManager.RollupData rollup, uint64 pendingStateNum, uint64 initNumBatch, uint64 finalNewBatch, bytes32 newLocalExitRoot, bytes32 newStateRoot, address beneficiary, bytes32[24] proof) internal Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollup | struct PolygonRollupManager.RollupData | Rollup Data storage pointer that will be used to the verification |pendingStateNum | uint64 | Init pending state, 0 if consolidated state is used |initNumBatch | uint64 | Batch which the aggregator starts the verification |finalNewBatch | uint64 | Last batch aggregator intends to verify |newLocalExitRoot | bytes32 | New local exit root once the batch is processed |newStateRoot | bytes32 | New State root once the batch is processed |beneficiary | address | Address that will receive the verification reward |proof | bytes32[24] | Fflonk proof tryConsolidatePendingState Internal function to consolidate the state automatically once sequence or verify batches are called. It tries to consolidate the first and the middle pending state in the queue. solidity function tryConsolidatePendingState( ) internal consolidatePendingState Consolidates any pending state that has already exceed the pendingStateTimeout. Can be called by the trusted aggregator, which can consolidate any state without the timeout restrictions. solidity function consolidatePendingState(uint32 rollupID, uint64 pendingStateNum) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollupID | uint32 | Rollup identifier |pendingStateNum | uint64 | Pending state to consolidate consolidatePendingState Internal function to consolidate any pending state that has already exceed the pendingStateTimeout. solidity function consolidatePendingState(struct PolygonRollupManager.RollupData rollup, uint64 pendingStateNum) internal Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollup | struct PolygonRollupManager.RollupData | Rollup data storage pointer |pendingStateNum | uint64 | Pending state to consolidate overridePendingState Allows the trusted aggregator to override the pending state if it is possible to prove a different state root given the same batches. solidity function overridePendingState(uint32 rollupID, uint64 initPendingStateNum, uint64 finalPendingStateNum, uint64 initNumBatch, uint64 finalNewBatch, bytes32 newLocalExitRoot, bytes32 newStateRoot, bytes32[24] proof) external Parameters | Name | Type | Description | | :--- | :--- | :-- | |rollupID | uint32 | Rollup identifier |initPendingStateNum | uint64 | Init pending state, 0 if consolidated state is used |finalPendingStateNum | uint64 | Final pending state, that will be used to compare with the newStateRoot |initNumBatch | uint64 | Batch which the aggregator starts the verification |finalNewBatch | uint64 | Last batch aggregator intends to verify |newLocalExitRoot | bytes32 | New local exit root once the batch is processed |newStateRoot | bytes32 | New State root once the batch is processed |proof | bytes32[24] | Fflonk proof proveNonDeterministicPendingState Activates the emergency state if its possible to prove a different state root given the same batches. solidity function proveNonDeterministicPendingState(uint32 rollupID, uint64 initPendingStateNum, uint64 finalPendingStateNum, uint64 initNumBatch, uint64 finalNewBatch, bytes32 newLocalExitRoot, bytes32 newStateRoot,

bytes32[24] proof) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollupID | uint32 | Rollup identifier |initPendingStateNum | uint64 | Init pending state, 0 if consolidated state is used |finalPendingStateNum | uint64 | Final pending state, that will be used to compare with the newStateRoot |initNumBatch | uint64 | Batch which the aggregator starts the verification |finalNewBatch | uint64 | Last batch aggregator intends to verify |newLocalExitRoot | bytes32 | New local exit root once the batch is processed |newStateRoot | bytes32 | New State root once the batch is processed |proof | bytes32[24] | Fflonk proof proveDistinctPendingState Internal function that proves a different state root given the same batches to verify. solidity function proveDistinctPendingState(struct PolygonRollupManager.RollupData rollup, uint64 initPendingStateNum, uint64 finalPendingStateNum, uint64 initNumBatch, uint64 finalNewBatch, bytes32 newLocalExitRoot, bytes32 newStateRoot, bytes32[24] proof) internal Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollup | struct PolygonRollupManager.RollupData | Rollup Data struct that will be checked |initPendingStateNum | uint64 | Init pending state, 0 if consolidated state is used |finalPendingStateNum | uint64 | Final pending state, that will be used to compare with the newStateRoot |initNumBatch | uint64 | Batch which the aggregator starts the verification |finalNewBatch | uint64 | Last batch aggregator intends to verify |newLocalExitRoot | bytes32 | New local exit root once the batch is processed |newStateRoot | bytes32 | New State root once the batch is processed |proof | bytes32[24] | Fflonk proof updateBatchFee Function to update the batch fee based on the new verified batches. The batch fee is not updated when the trusted aggregator verifies batches. solidity function updateBatchFee(struct PolygonRollupManager.RollupData newLastVerifiedBatch) internal Parameters | Name | Type | Description | | :--- | :--- | :--- | |newLastVerifiedBatch | struct PolygonRollupManager.RollupData | New last verified batch activateEmergencyState Function to activate emergency state, which also enables the emergency mode on both PolygonRollupManager and PolygonZkEVMBridge contracts. If not called by the owner, it must not have been aggregated within a HALTAGGREGATIONTIMEOUT period and an emergency state should not have happened in the same period. solidity function activateEmergencyState( ) external deactivateEmergencyState Function to deactivate emergency state on both PolygonRollupManager and PolygonZkEVMBridge contracts. solidity function deactivateEmergencyState( ) external activateEmergencyState Internal function to activate emergency state on both PolygonRollupManager and PolygonZkEVMBridge contracts. solidity function activateEmergencyState( ) internal setTrustedAggregatorTimeout Set a new pending state timeout. The timeout can only be lowered, except if emergency state is active. solidity function setTrustedAggregatorTimeout(uint64 newTrustedAggregatorTimeout) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |newTrustedAggregatorTimeout | uint64 | Trusted aggregator timeout setPendingStateTimeout Set a new trusted aggregator timeout. The timeout can only be lowered, except if emergency state is active. solidity function setPendingStateTimeout(uint64 newPendingStateTimeout) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |newPendingStateTimeout | uint64 | Trusted aggregator timeout setMultiplierBatchFee Set a new multiplier batch fee. solidity function setMultiplierBatchFee(uint16 newMultiplierBatchFee) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |newMultiplierBatchFee | uint16 | multiplier batch fee setVerifyBatchTimeTarget Set a new verify batch time target. This value will only be relevant once the aggregation is decentralized, so the trustedAggregatorTimeout should be zero or very close to zero. solidity function setVerifyBatchTimeTarget(uint64 newVerifyBatchTimeTarget) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |newVerifyBatchTimeTarget | uint64 | Verify batch time target setBatchFee solidity function setBatchFee(uint256 newBatchFee) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |newBatchFee | uint256 | new batch fee getRollupExitRoot Get the current rollup exit root: Compute the rollup exit root by using all the local exit roots of all rollups. solidity function getRollupExitRoot( ) public returns (bytes32) getLastVerifiedBatch solidity function getLastVerifiedBatch( ) public returns (uint64) getLastVerifiedBatch solidity function getLastVerifiedBatch( ) internal returns (uint64) isPendingStateConsolidable Returns a boolean that indicates if the pendingStateNum is consolidate-able. solidity function isPendingStateConsolidable(uint32 rollupID, uint64 pendingStateNum) public returns (bool) Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollupID | uint32 | Rollup id |pendingStateNum | uint64 | Pending state number to check !!! note - This function does not check if the pending state currently exists, or if it's consolidated already. isPendingStateConsolidable Returns a boolean that indicates if the pendingStateNum is consolidate-able. solidity function isPendingStateConsolidable(struct PolygonRollupManager.RollupData rollup, uint64 pendingStateNum) internal returns (bool) Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollup | struct PolygonRollupManager.RollupData | Rollup data storage pointer |pendingStateNum | uint64 | Pending state number to check !!! note - This function does not check if the pending state currently exists, or if it's consolidated already. calculateRewardPerBatch Function to calculate the reward to verify a single batch. solidity function calculateRewardPerBatch( ) public returns (uint256) getBatchFee This function is used instead of the automatic public view one. solidity function getBatchFee( ) public returns (uint256) getForcedBatchFee solidity function getForcedBatchFee( ) public returns (uint256) getInputSnarkBytes Function to calculate the input snark bytes. solidity function getInputSnarkBytes(uint32 rollupID, uint64 initNumBatch, uint64 finalNewBatch, bytes32 newLocalExitRoot, bytes32 oldStateRoot, bytes32 newStateRoot) public returns (bytes) Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollupID | uint32 | Rollup id used to calculate the input snark bytes |initNumBatch | uint64 | Batch which the aggregator starts the verification |finalNewBatch | uint64 | Last batch aggregator intends to verify |newLocalExitRoot | bytes32 | New local exit root once the batch is processed |oldStateRoot | bytes32 | State root before batch is processed |newStateRoot | bytes32 | New State root once the batch is processed getInputSnarkBytes Function to calculate the input snark bytes. solidity function getInputSnarkBytes(struct PolygonRollupManager.RollupData rollup, uint64 initNumBatch, uint64 finalNewBatch, bytes32 newLocalExitRoot, bytes32 oldStateRoot, bytes32 newStateRoot) internal returns (bytes) Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollup | struct PolygonRollupManager.RollupData | Rollup data storage pointer |initNumBatch | uint64 | Batch which the aggregator starts the verification |finalNewBatch | uint64 | Last batch aggregator intends to verify |newLocalExitRoot | bytes32 | New local exit root once the batch is processed |oldStateRoot | bytes32 | State root before batch is processed |newStateRoot | bytes32 | New State root once the batch is processed checkStateRootInsidePrime Function to check if the state root is inside of the prime field. solidity function checkStateRootInsidePrime(uint256 newStateRoot) internal returns (bool) Parameters | Name | Type | Description | | :--- | :--- | :--- | |newStateRoot | uint256 | New State root once the batch is processed getRollupBatchNumToStateRoot Get rollup state root given a batch number. solidity function getRollupBatchNumToStateRoot(uint32 rollupID, uint64 batchNum) public returns (bytes32) Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollupID | uint32 | Rollup identifier |batchNum | uint64 | Batch number getRollupSequencedBatches Get rollup sequence batches struct given a batch number. solidity function getRollupSequencedBatches(uint32 rollupID, uint64 batchNum) public returns (struct LegacyZKEVMStateVariables.SequencedBatchData) Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollupID | uint32 | Rollup identifier |batchNum | uint64 | Batch number getRollupPendingStateTransitions Get rollup sequence pending state struct given a batch number. solidity function getRollupPendingStateTransitions(uint32 rollupID, uint64 batchNum) public returns (struct LegacyZKEVMStateVariables.PendingState) Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollupID | uint32 | Rollup identifier |batchNum | uint64 | Batch number Events AddNewRollupType Emitted when a new rollup type is added. solidity event AddNewRollupType( ) ObsoleteRollupType Emitted when a a rollup type is obsoleted. solidity event ObsoleteRollupType( ) CreateNewRollup Emitted when a new rollup is created based on a rollupType. solidity event CreateNewRollup( ) AddExistingRollup Emitted when an existing rollup is added. solidity event AddExistingRollup( ) UpdateRollup Emitted when a rollup is updated. solidity event UpdateRollup( ) OnSequenceBatches Emitted when a new verifier is added. solidity event OnSequenceBatches( ) VerifyBatches Emitted when an aggregator verifies batches. solidity event VerifyBatches( ) VerifyBatchesTrustedAggregator Emitted when the trusted aggregator verifies batches. solidity event VerifyBatchesTrustedAggregator( ) ConsolidatePendingState Emitted when pending state is consolidated. solidity event ConsolidatePendingState( ) ProveNonDeterministicPendingState Emitted when is proved a different state given the same batches. solidity event ProveNonDeterministicPendingState( ) OverridePendingState Emitted when the trusted aggregator overrides pending state. solidity event OverridePendingState( ) SetTrustedAggregatorTimeout Emitted when is updated the trusted aggregator timeout. solidity event SetTrustedAggregatorTimeout( ) SetPendingStateTimeout Emitted when is updated the pending state timeout. solidity event SetPendingStateTimeout( ) SetMultiplierBatchFee Emitted when is updated the multiplier batch fee. solidity event SetMultiplierBatchFee( ) SetVerifyBatchTimeTarget Emitted when is updated the verify batch timeout solidity event SetVerifyBatchTimeTarget( ) SetTrustedAggregator Emitted when is updated the trusted aggregator address. solidity event SetTrustedAggregator( ) SetBatchFee Emitted when is updated the batch fee. solidity event SetBatchFee( ) # PolygonZkEVMBridgeV2.md: Bridge contract deployed on Ethereum and all Polygon rollups. Responsible for managing the token interactions with other networks. Functions constructor Disable initializers on the implementation following best practices. solidity function constructor( ) public initialize The value of polygonRollupManager on the L2 deployment of the contract is address(0), so an emergency state is not possible for the L2 deployment of the bridge. solidity function initialize(uint32 networkID, address gasTokenAddress, uint32 gasTokenNetwork, contract IBasePolygonZkEVMGlobalExitRoot globalExitRootManager, address polygonRollupManager, bytes gasTokenMetadata) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |networkID | uint32 | networkID |gasTokenAddress | address | gas token address |gasTokenNetwork | uint32 | gas token network |globalExitRootManager | contract IBasePolygonZkEVMGlobalExitRoot | global exit root manager address |polygonRollupManager | address | polygonZkEVM address |gasTokenMetadata | bytes | Abi encoded gas token metadata bridgeAsset Deposit add a new leaf to the Merkle tree. solidity function bridgeAsset(uint32 destinationNetwork, address destinationAddress, uint256 amount, address token, bool forceUpdateGlobalExitRoot, bytes permitData) public !!! note - If this function is called with a reentrant token, it is possible to claimTokens in the same call, thus reducing the supply of tokens on this contract, and actually locking tokens in the contract. - Therefore we recommend to third-party bridges that if they do implement reentrant beforeTransfer for some reentrant tokens, not to call any external address. !!! note - User/UI must be aware of the existing/available networks when choosing the destination network. Parameters | Name | Type | Description | | :--- | :--- | :--- | |destinationNetwork | uint32 | Network destination |destinationAddress | address | Address destination |amount | uint256 | Amount of tokens |token | address | Token address, 0 address is reserved for ether |forceUpdateGlobalExitRoot | bool | Indicates if the new global exit root is updated or not |permitData | bytes | Raw data of the call permit of the token bridgeMessage Bridge message and send ETH value. solidity function bridgeMessage(uint32 destinationNetwork, address destinationAddress, bool forceUpdateGlobalExitRoot, bytes metadata) external !!! note - User/UI must be aware of the existing/available networks when choosing the destination network. Parameters | Name | Type | Description | | :--- | :--- | :--- | |destinationNetwork | uint32 | Network destination |destinationAddress | address | Address destination |forceUpdateGlobalExitRoot | bool | Indicates if the new global exit root is updated or not |metadata | bytes | Message metadata bridgeMessageWETH Bridge message and send ETH value. solidity function bridgeMessageWETH(uint32 destinationNetwork, address destinationAddress, uint256 amountWETH, bool forceUpdateGlobalExitRoot, bytes metadata) external !!! note - User/UI must be aware of the existing/available networks when choosing the destination network Parameters | Name | Type | Description | | :--- | :--- | :--- | |destinationNetwork | uint32 | Network destination |destinationAddress | address | Address destination |amountWETH | uint256 | Amount of WETH tokens |forceUpdateGlobalExitRoot | bool | Indicates if the new global exit root is updated or not |metadata | bytes | Message metadata bridgeMessage Bridge message and send ETH value. solidity function bridgeMessage(uint32 destinationNetwork, address destinationAddress, uint256 amountEther, bool forceUpdateGlobalExitRoot, bytes metadata) internal Parameters | Name | Type | Description | | :--- | :--- | :--- | |destinationNetwork | uint32 | Network destination |destinationAddress | address | Address destination |amountEther | uint256 | Amount of ether along with the message |forceUpdateGlobalExitRoot | bool | Indicates if the new global exit root is updated or not |metadata | bytes | Message metadata claimAsset Verify merkle proof and withdraw tokens/ETH. solidity function claimAsset(bytes32[32] smtProofLocalExitRoot, bytes32[32] smtProofRollupExitRoot, uint256 globalIndex, bytes32 mainnetExitRoot, bytes32 rollupExitRoot, uint32 originNetwork, address originTokenAddress, uint32 destinationNetwork, address destinationAddress, uint256 amount, bytes metadata) external Parameters s | Name | Type | Description | | :--- | :--- | :--- | |smtProofLocalExitRoot | bytes32[32] | Smt proof to proof the leaf against the network exit root |smtProofRollupExitRoot | bytes32[32] | Smt proof to proof the rollupLocalExitRoot against the rollups exit root |globalIndex | uint256 | Global index is defined as: | 191 bits | 1 bit | 32 bits | 32 bits | | 0 | mainnetFlag | rollupIndex | localRootIndex | note that only the rollup index will be used only in case the mainnet flag is 0 note that global index do not assert the unused bits to 0. This means that when synching the events, the globalIndex must be decoded the same way that in the Smart contract to avoid possible synch attacks |mainnetExitRoot | bytes32 | Mainnet exit root |rollupExitRoot | bytes32 | Rollup exit root |originNetwork | uint32 | Origin network |originTokenAddress | address | Origin token address, 0 address is reserved for ether |destinationNetwork | uint32 | Network destination |destinationAddress | address | Address destination |amount | uint256 | Amount of tokens |metadata | bytes | Abi encoded metadata if any, empty otherwise claimMessage Verify merkle proof and execute message. If the receiving address is an EOA, the call results as a success which means that the amount of ether transfers correctly, but the message does not trigger any execution. solidity function claimMessage(bytes32[32] smtProofLocalExitRoot, bytes32[32] smtProofRollupExitRoot, uint256 globalIndex, bytes32 mainnetExitRoot, bytes32 rollupExitRoot, uint32 originNetwork, address originAddress, uint32 destinationNetwork, address destinationAddress, uint256 amount, bytes metadata) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |smtProofLocalExitRoot | bytes32[32] | Smt proof to proof the leaf against the exit root |smtProofRollupExitRoot | bytes32[32] | Smt proof to proof the rollupLocalExitRoot against the rollups exit root |globalIndex | uint256 | Global index is defined as: | 191 bits | 1 bit | 32 bits | 32 bits | | 0 | mainnetFlag | rollupIndex | localRootIndex | note that only the rollup index will be used only in case the mainnet flag is 0 note that global index do not assert the unused bits to 0. This means that when synching the events, the globalIndex must be decoded the same way that in the Smart contract to avoid possible synch attacks |mainnetExitRoot | bytes32 | Mainnet exit root |rollupExitRoot | bytes32 | Rollup exit root |originNetwork | uint32 | Origin network |originAddress | address | Origin address |destinationNetwork | uint32 | Network destination |destinationAddress | address | Address destination |amount | uint256 | message value |metadata | bytes | Abi encoded metadata if any, empty otherwise precalculatedWrapperAddress Returns the precalculated address of a wrapper using the token information. solidity function precalculatedWrapperAddress(uint32 originNetwork, address originTokenAddress, string name, string symbol, uint8 decimals) public returns (address) !!! note - Updating the metadata of a token is not supported. - Since the metadata has relevance in the address deployed, this function does not return a valid wrapped address if the metadata provided is not the original one. Parameters | Name | Type | Description | | :--- | :--- | :--- | |originNetwork | uint32 | Origin network |originTokenAddress | address | Origin token address, 0 address is reserved for ether |name | string | Name of the token |symbol | string | Symbol of the token |decimals | uint8 | Decimals of the token getTokenWrappedAddress Returns the address of a wrapper using the token information if already exist. solidity function getTokenWrappedAddress(uint32 originNetwork, address originTokenAddress) external returns (address) Parameters | Name | Type | Description | | :--- | :--- | :--- | |originNetwork | uint32 | Origin network |originTokenAddress | address | Origin token address, 0 address is reserved for ether activateEmergencyState Function to activate the emergency state. Can only be called by the Polygon ZK-EVM in extreme situations. solidity function activateEmergencyState( ) external deactivateEmergencyState Function to deactivate the emergency state. Can only be called by the Polygon ZK-EVM. solidity function deactivateEmergencyState( ) external verifyLeaf Verify leaf and checks that it has not been claimed. solidity function verifyLeaf(bytes32[32] smtProofLocalExitRoot, bytes32[32] smtProofRollupExitRoot, uint256 globalIndex, bytes32 mainnetExitRoot, bytes32 rollupExitRoot, bytes32 leafValue) internal Parameters | Name | Type | Description | | :--- | :--- | :--- | |smtProofLocalExitRoot | bytes32[32] | Smt proof |smtProofRollupExitRoot | bytes32[32] | Smt proof |globalIndex | uint256 | Index of the leaf |mainnetExitRoot | bytes32 | Mainnet exit root |rollupExitRoot | bytes32 | Rollup exit root |leafValue | bytes32 | leaf value isClaimed Function to check if an index is claimed or not. solidity function isClaimed(uint32 leafIndex, uint32 sourceBridgeNetwork) external returns (bool) Parameters | Name | Type | Description | | :--- | :--- | :--- | |leafIndex | uint32 | Index |sourceBridgeNetwork | uint32 | Origin network updateGlobalExitRoot Function to update the globalExitRoot if the last deposit is not submitted. solidity function updateGlobalExitRoot( ) external updateGlobalExitRoot Function to update the globalExitRoot. solidity function updateGlobalExitRoot( ) internal permit Function to call token permit method of extended ERC20 + @param token ERC20 token address. solidity function permit(address amount, uint256 permitData) internal Parameters | Name | Type | Description | | :--- | :--- | :--- | |amount | address Quantity that is expected to be allowed |permitData | uint256 | Raw data of the call permit of the token deployWrappedToken Internal function that uses create2 to deploy the wrapped tokens. solidity function deployWrappedToken(bytes32 salt, bytes constructorArgs) internal returns (contract TokenWrapped newWrappedToken) Parameters | Name | Type | Description | | :--- | :--- | :--- | |salt | bytes32 | Salt used in create2 params, tokenInfoHash will be used as salt for all wrappeds except for bridge native WETH, that will be bytes32(0) |constructorArgs | bytes | Encoded constructor args for the wrapped token safeSymbol Provides a safe ERC20.symbol version which returns NOSYMBOL as fallback string. solidity function safeSymbol(address token) internal returns (string) Parameters | Name | Type | Description | | :--- | :--- | :--- | |token | address | The address of the ERC-20 token contract safeName Provides a safe ERC20.name version which returns NONAME as fallback string. solidity function safeName(address token) internal returns (string) Parameters | Name | Type | Description | | :--- | :--- | :--- | |token | address | The address of the ERC-20 token contract. safeDecimals Provides a safe ERC20.decimals version which returns 18 as fallback value. solidity function safeDecimals(address token) internal returns (uint8) !!! warn - Tokens with (decimals > 255) are not supported. Parameters | Name | Type | Description | | :--- | :--- | :--- | |token | address | The address of the ERC-20 token contract returnDataToString Function to convert returned data to string; returns NOTVALIDENCODING as fallback value. solidity function returnDataToString(bytes data) internal returns (string) Parameters | Name | Type | Description | | :--- | :--- | :--- | |data | bytes | returned data getTokenMetadata Returns the encoded token metadata. solidity function getTokenMetadata(address token) public returns (bytes) Parameters | Name | Type | Description | | :--- | :--- | :--- | |token | address | Address of the token calculateTokenWrapperAddress Returns the precalculated address of a wrapper using the token address. solidity function calculateTokenWrapperAddress(uint32 originNetwork, address originTokenAddress, address token) external returns (address) !!! note - Updating the metadata of a token is not supported. - Since the metadata has relevance in the address deployed, this function does not return a valid wrapped address if the metadata provided is not the original one. Parameters | Name | Type | Description | | :--- | :--- | :--- | |originNetwork | uint32 | Origin network |originTokenAddress | address | Origin token address, 0 address is reserved for ether |token | address | Address of the token to calculate the wrapper address Events BridgeEvent Emitted when bridge assets or messages to another network. solidity event BridgeEvent( ) ClaimEvent Emitted when a claim is done from another network. solidity event ClaimEvent( ) NewWrappedToken

Emitted when a new wrapped token is created. solidity event NewWrappedToken( ) # PolygonZkEVMGlobalExitRootV2.md: Contract responsible for managing the exit roots across multiple networks. Functions constructor solidity function constructor(address rollupManager, address bridgeAddress) public Parameters | Name | Type | Description | | :--- | :--- | :--- | |rollupManager | address | Rollup manager contract address |bridgeAddress | address | PolygonZkEVMBridge contract address updateExitRoot Updates the exit root of any of the networks and the global exit root. solidity function updateExitRoot(bytes32 newRoot) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |newRoot | bytes32 | new exit tree root getLastGlobalExitRoot Returns last global exit root. solidity function getLastGlobalExitRoot( ) public returns (bytes32) getRoot Computes and returns the Merkle root of the L1InfoTree. solidity function getRoot( ) public returns (bytes32) getLeafValue Given leaf data, it returns the leaf hash. solidity function getLeafValue(bytes32 newGlobalExitRoot, uint256 lastBlockHash, uint64 timestamp) public returns (bytes32) Parameters | Name | Type | Description | | :--- | :--- | :--- | |newGlobalExitRoot | bytes32 | Last global exit root |lastBlockHash | uint256 | Last accesible block hash |timestamp | uint64 | Ethereum timestamp in seconds Events UpdateL1InfoTree Emitted when the global exit root is updated. solidity event UpdateL1InfoTree( ) # CDKDataCommittee.md: Functions initialize solidity function initialize( ) external setupCommittee Allows the admin to setup members of the committee. solidity function setupCommittee(uint256 requiredAmountOfSignatures, string[] urls, bytes addrsBytes) external !!! note - The system requires N/M signatures where N => requiredAmountOfSignatures and M => urls.length. - The number of urls must be the same as the addresses encoded in the addrsBytes. - A member is represented by a url and the address contained in urls[i] and addrsBytes[iADDRSIZE : iADDRSIZE + ADDRSIZE]. Parameters | Name | Type | Description | | :--- | :--- | :--- | |requiredAmountOfSignatures | uint256 | Required amount of signatures |urls | string[] | List of urls of the members of the committee |addrsBytes | bytes | Byte array that contains the addresses of the members of the committee getAmountOfMembers solidity function getAmountOfMembers( ) public returns (uint256) verifySignatures Verifies that the given signedHash has been signed by requiredAmountOfSignatures committee members. solidity function verifySignatures(bytes32 signedHash, bytes signaturesAndAddrs) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |signedHash | bytes32 | Hash that must have been signed by requiredAmountOfSignatures of committee members |signaturesAndAddrs | bytes | Byte array containing the signatures and all the addresses of the committee in ascending order [signature 0, ..., signature requiredAmountOfSignatures -1, address 0, ... address N] note that each ECDSA signatures are used, therefore each one must be 65 bytes Events CommitteeUpdated Emitted when the committee is updated. solidity event CommitteeUpdated(bytes32 committeeHash) Parameters | Name | Type | Description | | :----------------------------- | :------------ | :--- | |committeeHash| bytes32 | hash of the addresses of the committee members | # PolygonDataComittee.md: Functions constructor solidity function constructor(contract IPolygonZkEVMGlobalExitRoot globalExitRootManager, contract IERC20Upgradeable pol, contract IPolygonZkEVMBridge bridgeAddress, contract PolygonRollupManager rollupManager) public Parameters | Name | Type | Description | | :--- | :--- | :--- | |globalExitRootManager | contract IPolygonZkEVMGlobalExitRoot | Global exit root manager address |pol | contract IERC20Upgradeable | POL token address |bridgeAddress | contract IPolygonZkEVMBridge | Bridge address |rollupManager | contract PolygonRollupManager | Global exit root manager address sequenceBatches solidity function sequenceBatches( ) public sequenceBatchesDataCommittee Allows a sequencer to send multiple batches. solidity function sequenceBatchesDataCommittee(struct PolygonDataComittee.ValidiumBatchData[] batches, address l2Coinbase, bytes dataAvailabilityMessage) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |batches | struct PolygonDataComittee.ValidiumBatchData[] | Struct array which holds the necessary data to append new batches to a sequence |l2Coinbase | address | Address that will receive the fees from L2 |dataAvailabilityMessage | bytes | Byte array containing signatures and all addresses of the committee members in the ascending order [signature 0, ..., signature requiredAmountOfSignatures -1, address 0, ... address N] note that each ECDSA signatures are used, therefore each one must be 65 bytes switchSequenceWithDataAvailability Allows the admin to activate force batches. This action is not reversible. solidity function switchSequenceWithDataAvailability( ) external Events SwitchSequenceWithDataAvailability solidity event SwitchSequenceWithDataAvailability( ) Emitted when switching the sequencing functionality. # PolygonDataCommittee.md: Functions constructor Disables initializers on the implementation, following best practices. solidity function constructor( ) public initialize solidity function initialize( ) external setupCommittee Allows the admin to setup the members of the committee. !!! note - The system requires N/M signatures where N => requiredAmountOfSignatures and M => urls.length. - The number of urls must be the same as addresses encoded in the addrsBytes. - A member is represented by a url and the address contained in urls[i] and addrsBytes. [iADDRSIZE : iADDRSIZE + ADDRSIZE]. solidity function setupCommittee(uint256 requiredAmountOfSignatures, string[] urls, bytes addrsBytes) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |requiredAmountOfSignatures | uint256 | Required amount of signatures |urls | string[] | List of urls of the members of the committee |addrsBytes | bytes | Byte array that contains the addressess of the members of the committee verifyMessage Verifies that the given signedHash has been signed by the requiredAmountOfSignatures committee members. solidity function verifyMessage(bytes32 signedHash, bytes signaturesAndAddrs) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |signedHash | bytes32 | Hash that must have been signed by the requiredAmountOfSignatures of committee members. | |signaturesAndAddrs | bytes | Byte array containing signatures and all addresses of the committee members in ascending order [signature 0, ..., signature requiredAmountOfSignatures -1, address 0, ... address N].
Note that all signatures are ECDSA, therefore each must be 65 bytes long. | getAmountOfMembers Returns the number of committee members. solidity function getAmountOfMembers( ) public returns (uint256) getProcotolName Returns the protocol name. solidity function getProcotolName( ) external returns (string) Events CommitteeUpdated Emitted when the committee is updated. solidity event CommitteeUpdated(bytes32 committeeHash) Parameters | Name | Type | Description | | :----------------------------- | :------------ | :--- | |committeeHash| bytes32 | hash of the addresses of the committee members | # PolygonValidiumEtrog.md: Functions constructor solidity function constructor(contract IPolygonZkEVMGlobalExitRootV2 globalExitRootManager, contract IERC20Upgradeable pol, contract IPolygonZkEVMBridgeV2 bridgeAddress, contract PolygonRollupManager rollupManager) public Parameters | Name | Type | Description | | :--- | :--- | :--- | |globalExitRootManager | contract IPolygonZkEVMGlobalExitRootV2 | Global exit root manager address |pol | contract IERC20Upgradeable | POL token address |bridgeAddress | contract IPolygonZkEVMBridgeV2 | Bridge address |rollupManager | contract PolygonRollupManager | Global exit root manager address sequenceBatchesValidium Allows a sequencer to send multiple batches. solidity function sequenceBatchesValidium(struct PolygonValidiumEtrog.ValidiumBatchData[] batches, uint64 maxSequenceTimestamp, uint64 initSequencedBatch, address l2Coinbase, bytes dataAvailabilityMessage) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |batches | struct PolygonValidiumEtrog.ValidiumBatchData[] | Struct array which holds data necessary for appending new batches to the sequence |maxSequenceTimestamp | uint64 | Max timestamp of the sequence.
This timestamp must be within a safety range (actual + 36 seconds).
This timestamp should be equal or greater than that of the last block inside the sequence, otherwise this batch is invalidated by the circuit. | |initSequencedBatch | uint64 | This parameter must match the current last batch sequenced.
This is a protection mechanism against the sequencer sending undesired data. | |l2Coinbase | address | Address that will receive the fees from L2 | |dataAvailabilityMessage | bytes | Byte array containing signatures and all addresses of the committee members in ascending order
[signature 0, ..., signature requiredAmountOfSignatures -1, address 0, ... address N].
Note that all signatures are ECDSA, therefore each must be 65 bytes long. | Note that POL is not a reentrant token. sequenceBatches Allows a sequencer to send multiple batches. solidity function sequenceBatches(struct PolygonRollupBaseEtrog.BatchData[] batches, uint64 maxSequenceTimestamp, uint64 initSequencedBatch, address l2Coinbase) public Parameters | Name | Type | Description | | :--- | :--- | :--- | |batches | struct PolygonRollupBaseEtrog.BatchData[] | Struct array which holds data necessary for appending new batches to the sequence. | |maxSequenceTimestamp | uint64 | Max timestamp of the sequence. This timestamp must be within a safety range (actual + 36 seconds). It should be equal or greater than the last block inside the sequence, otherwise this batch is invalidated by circuit. | |initSequencedBatch | uint64 | This parameter must match the current last batch sequenced. This is a protection mechanism against the sequencer sending undesired data. | |l2Coinbase | address | Address that will receive the fees from L2.
Note that POL is not a reentrant token. | setDataAvailabilityProtocol Allows the admin to set a new data availability protocol. solidity function setDataAvailabilityProtocol(contract IDataAvailabilityProtocol newDataAvailabilityProtocol) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |newDataAvailabilityProtocol | contract IDataAvailabilityProtocol | Address of the new data availability protocol. | switchSequenceWithDataAvailability Allows the admin to switch the sequencing functionality. solidity function switchSequenceWithDataAvailability(bool newIsSequenceWithDataAvailabilityAllowed) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |newIsSequenceWithDataAvailabilityAllowed | bool | A Boolean switch sequencing mode. | Events SetDataAvailabilityProtocol Emitted when the admin updates the data availability protocol. solidity event SetDataAvailabilityProtocol( ) SwitchSequenceWithDataAvailability Emitted when switching the sequencing functionality to a validium, a network with a data availability committee. solidity event SwitchSequenceWithDataAvailability( ) # PolygonRollupBaseEtrogNoGap.md: Functions constructor solidity function constructor(contract IPolygonZkEVMGlobalExitRootV2 globalExitRootManager, contract IERC20Upgradeable pol, contract IPolygonZkEVMBridgeV2 bridgeAddress, contract PolygonRollupManager rollupManager) public Parameters | Name | Type | Description | | :--- | :--- | :--- | |globalExitRootManager | contract IPolygonZkEVMGlobalExitRootV2 | Global exit root manager address. | |pol | contract IERC20Upgradeable | POL token address. | |bridgeAddress | contract IPolygonZkEVMBridgeV2 | Bridge address. | |rollupManager | contract PolygonRollupManager | Global exit root manager address. | initializeMigration solidity function initializeMigration( ) external sequenceBatchesValidium Allows a sequencer to send multiple batches. solidity function sequenceBatchesValidium(struct PolygonValidiumStorageMigration.ValidiumBatchData[] batches, uint64 maxSequenceTimestamp, uint64 initSequencedBatch, address l2Coinbase, bytes dataAvailabilityMessage) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |batches | struct PolygonValidiumStorageMigration.ValidiumBatchData[] | Struct array which holds the data necessary for appending new batches to the sequence. | |maxSequenceTimestamp | uint64 | Max timestamp of the sequence.
This timestamp must be within a safety range (actual + 36 seconds).
It should be equal or greater than the last block inside the sequence. Otherwise the batch is invalidated by the circuit. | |initSequencedBatch | uint64 | This parameter must match the current last batch sequenced.
This is a protection mechanism against the sequencer sending undesired data. | |l2Coinbase | address | Address that will receive the fees from L2. | |dataAvailabilityMessage | bytes | Byte array containing the signatures and all the addresses of the committee in ascending order
[signature 0, ..., signature requiredAmountOfSignatures -1, address 0, ... address N]
Note that al signatures are ECDSA, therefore each must be 65 bytes long. | Note that POL is not a reentrant token. sequenceBatches Allows a sequencer to send multiple batches. solidity function sequenceBatches(struct PolygonRollupBaseEtrogNoGap.BatchData[] batches, uint64 maxSequenceTimestamp, uint64 initSequencedBatch, address l2Coinbase) public Parameters | Name | Type | Description | | :--- | :--- | :--- | |batches | struct PolygonRollupBaseEtrogNoGap.BatchData[] | Struct array which holds the data necessary for appending new batches to the sequence. | |maxSequenceTimestamp | uint64 | Max timestamp of the sequence.
The timestamp must be within a safety range (actual + 36 seconds).
It should be equal or greater than the last block inside the sequence. Otherwise the batch is invalidated by the circuit. | |initSequencedBatch | uint64 | This parameter must match the current last batch sequenced.
This is a protection mechanism against the sequencer sending undesired data. | |l2Coinbase | address | Address that will receive the fees from L2.
Note that POL is not a reentrant token. | setDataAvailabilityProtocol Allows the admin to set a new data availability protocol. solidity function setDataAvailabilityProtocol(contract IDataAvailabilityProtocol newDataAvailabilityProtocol) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |newDataAvailabilityProtocol | contract IDataAvailabilityProtocol | Address of the new data availability protocol. | switchSequenceWithDataAvailability Allows the admin to switch the sequencing functionality to a data availability configuration. solidity function switchSequenceWithDataAvailability(bool newIsSequenceWithDataAvailabilityAllowed) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |newIsSequenceWithDataAvailabilityAllowed | bool | Boolean to switch. | Events SetDataAvailabilityProtocol Emitted when the admin updates the data availability protocol. solidity event SetDataAvailabilityProtocol( ) SwitchSequenceWithDataAvailability Emitted when switching the sequencing functionality to a data availability configuration. solidity event SwitchSequenceWithDataAvailability( ) # PolygonValidiumStorageMigration.md: Functions constructor solidity function constructor(contract IPolygonZkEVMGlobalExitRootV2 globalExitRootManager, contract IERC20Upgradeable pol, contract IPolygonZkEVMBridgeV2 bridgeAddress, contract PolygonRollupManager rollupManager) public Parameters | Name | Type | Description | | :--- | :--- | :--- | |globalExitRootManager | contract IPolygonZkEVMGlobalExitRootV2 | Global exit root manager address. | |pol | contract IERC20Upgradeable | POL token address. | |bridgeAddress | contract IPolygonZkEVMBridgeV2 | Bridge address. | |rollupManager | contract PolygonRollupManager | Global exit root manager address. | initializeMigration solidity function initializeMigration( ) external sequenceBatchesValidium Allows a sequencer to send multiple batches. solidity function sequenceBatchesValidium(struct PolygonValidiumStorageMigration.ValidiumBatchData[] batches, uint64 maxSequenceTimestamp, uint64 initSequencedBatch, address l2Coinbase, bytes dataAvailabilityMessage) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |batches | struct PolygonValidiumStorageMigration.ValidiumBatchData[] | Struct array which holds the data necessary for appending new batches to the sequence. | |maxSequenceTimestamp | uint64 | Max timestamp of the sequence.
This timestamp must be within a safety range (actual + 36 seconds).
It should be equal or greater than the last block inside the sequence. Otherwise the batch is invalidated by the circuit. | |initSequencedBatch | uint64 | This parameter must match the current last batch sequenced.
This is a protection mechanism against the sequencer sending undesired data. | |l2Coinbase | address | Address that will receive the fees from L2. | |dataAvailabilityMessage | bytes | Byte array containing the signatures and all the addresses of the committee in ascending order
[signature 0, ..., signature requiredAmountOfSignatures -1, address 0, ... address N]
Note that al signatures are ECDSA, therefore each must be 65 bytes long. | Note that POL is not a reentrant token. sequenceBatches Allows a sequencer to send multiple batches. solidity function sequenceBatches(struct PolygonRollupBaseEtrogNoGap.BatchData[] batches, uint64 maxSequenceTimestamp, uint64 initSequencedBatch, address l2Coinbase) public Parameters | Name | Type | Description | | :--- | :--- | :--- | |batches | struct PolygonRollupBaseEtrogNoGap.BatchData[] | Struct array which holds the data necessary for appending new batches to the sequence. | |maxSequenceTimestamp | uint64 | Max timestamp of the sequence.
The timestamp must be within a safety range (actual + 36 seconds).
It should be equal or greater than the last block inside the sequence. Otherwise the batch is invalidated by the circuit. | |initSequencedBatch | uint64 | This parameter must match the current last batch sequenced.
This is a protection mechanism against the sequencer sending undesired data. | |l2Coinbase | address | Address that will receive the fees from L2.
Note that POL is not a reentrant token. | setDataAvailabilityProtocol Allows the admin to set a new data availability protocol. solidity function setDataAvailabilityProtocol(contract IDataAvailabilityProtocol newDataAvailabilityProtocol) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |newDataAvailabilityProtocol | contract IDataAvailabilityProtocol | Address of the new data availability protocol. | switchSequenceWithDataAvailability Allows the admin to switch the sequencing functionality to a data availability configuration. solidity function switchSequenceWithDataAvailability(bool newIsSequenceWithDataAvailabilityAllowed) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |newIsSequenceWithDataAvailabilityAllowed | bool | Boolean to switch. | Events SetDataAvailabilityProtocol Emitted when the admin updates the data availability protocol. solidity event SetDataAvailabilityProtocol( ) SwitchSequenceWithDataAvailability Emitted when switching the sequencing functionality to a data availability configuration. solidity event SwitchSequenceWithDataAvailability( ) # PolygonZkEVMEtrog.md: Functions constructor solidity function constructor(contract IPolygonZkEVMGlobalExitRootV2 globalExitRootManager, contract IERC20Upgradeable pol, contract IPolygonZkEVMBridgeV2 bridgeAddress, contract PolygonRollupManager rollupManager) public Parameters | Name | Type | Description | | :--- | :--- | :--- | |globalExitRootManager | contract IPolygonZkEVMGlobalExitRootV2 | Global exit root manager address. | |pol | contract IERC20Upgradeable | POL token address. | |bridgeAddress | contract IPolygonZkEVMBridgeV2 | Bridge address. | |rollupManager | contract PolygonRollupManager | Global exit root manager address. | # PolygonZkEVMExistentEtrog.md: Functions constructor solidity function constructor(contract IPolygonZkEVMGlobalExitRootV2 globalExitRootManager, contract IERC20Upgradeable pol, contract IPolygonZkEVMBridgeV2 bridgeAddress, contract PolygonRollupManager rollupManager) public Parameters | Name | Type | Description | | :--- | :--- | :--- | |globalExitRootManager | contract IPolygonZkEVMGlobalExitRootV2 | Global exit root manager address. | |pol | contract IERC20Upgradeable | POL token address. | |bridgeAddress | contract IPolygonZkEVMBridgeV2 | Bridge address. | |rollupManager | contract PolygonRollupManager | Global exit root manager address. | initializeUpgrade solidity function initializeUpgrade(address admin, address trustedSequencer, string trustedSequencerURL, string networkName, bytes32 lastAccInputHash) external !!!

note - This initializer is called instead of the PolygonRollupBase. - This is a special initializer since the zkEVM network has already been created. Parameters | Name | Type | Description | | :--- | :--- | :--- | |admin | address | Admin address. | |trustedSequencer | address | Trusted sequencer address. | |trustedSequencerURL | string | Trusted sequencer URL. | |networkName | string | L2 network name. | |lastAccInputHash | bytes32 | Acc input hash. | Events UpdateEtrogSequence Emitted when the system is updated to Etrog upgrade using this contract, and contains the etrog setup transaction. solidity event UpdateEtrogSequence( ) # PolygonZkEVMV2.md: Functions constructor solidity function constructor(contract IPolygonZkEVMGlobalExitRoot globalExitRootManager, contract IERC20Upgradeable pol, contract IPolygonZkEVMBridge bridgeAddress, contract PolygonRollupManager rollupManager) public Parameters | Name | Type | Description | | :--- | :--- | :--- | |globalExitRootManage | contract IPolygonZkEVMGlobalExitRoot | Global exit root manager address. | |pol | contract IERC20Upgradeable | POL token address. | |bridgeAddress | contract IPolygonZkEVMBridge | Bridge address. | |rollupManager | contract PolygonRollupManager | Global exit root manager address. | # PolygonZkEVMV2Existent.md: Functions constructor solidity function constructor(contract IPolygonZkEVMGlobalExitRoot globalExitRootManager, contract IERC20Upgradeable pol, contract IPolygonZkEVMBridge bridgeAddress, contract PolygonRollupManager rollupManager) public Parameters | Name | Type | Description | | :--- | :--- | :--- | |globalExitRootManager | contract IPolygonZkEVMGlobalExitRoot | Global exit root manager address. | |pol | contract IERC20Upgradeable | POL token address. | |bridgeAddress | contract IPolygonZkEVMBridge | Bridge address. | |rollupManager | contract PolygonRollupManager | Global exit root manager address. | # DepositContractBase.md: This contract is a helper for all functions related to the sparse Merkle tree. And it is based on the implementation of the deposit eth2.0 contract https://github.com/ethereum/consensus-specs/blob/dev/soliditydepositcontract/depositcontract.sol. Functions getRoot Computes and returns the Merkle root. solidity function getRoot( ) public returns (bytes32) addLeaf Adds a new leaf to the merkle tree. solidity function addLeaf(bytes32 leaf) internal Parameters | Name | Type | Description | | :--- | :--- | :--- | |leaf | bytes32 | Leaf | verifyMerkleProof Verifies a Merkle proof. solidity function verifyMerkleProof(bytes32 leafHash, bytes32[32] smtProof, uint32 index, bytes32 root) public returns (bool) Parameters | Name | Type | Description | | :--- | :--- | :--- | |leafHash | bytes32 | Leaf hash. | |smtProof | bytes32[32] | Smt proof. | |index | uint32 | Index of the leaf. | |root | bytes32 | Merkle root. | calculateRoot Calculates root via a Merkle proof. solidity function calculateRoot(bytes32 leafHash, bytes32[32] smtProof, uint32 index) public returns (bytes32) Parameters | Name | Type | Description | | :--- | :--- | :--- | |leafHash | bytes32 | Leaf hash. | |smtProof | bytes32[32] | Smt proof. | |index | uint32 | Index of the leaf. | # DepositContractLib.md: This contract is a helper for all functions related to the sparse Merkle tree. And it is based on the implementation of the deposit eth2.0 contract https://github.com/ethereum/consensus-specs/blob/dev/soliditydepositcontract/depositcontract.sol. Functions getRoot Computes and returns the Merkle root. solidity function getRoot( ) public returns (bytes32) addLeaf Adds a new leaf to a merkle tree. solidity function addLeaf(bytes32 leaf) internal Parameters | Name | Type | Description | | :--- | :--- | :--- | |leaf | bytes32 | Leaf. | verifyMerkleProof Verifies a Merkle proof. solidity functi verifyMerkleProof(bytes32 leaf, bytes32[32] smtProof, uint32 index, bytes32 root) public returns (bool) Parameters | Name | Type | Description | | :--- | :--- | :--- | |leaf | bytes32 | Leaf. | |smtProof | bytes32[32] | Smt proof. | |index | uint32 | Index of the leaf. | |root | bytes32 | Merkle root. | # DepositContractV2.md: This contract is used by the PolygonZkEVMBridge contract. It inherits the DepositContractBase and adds the logic to calculate the leaf of the exit tree. Functions getLeafValue Given the leaf data it returns the leaf value. solidity function getLeafValue(uint8 leafType, uint32 originNetwork, address originAddress, uint32 destinationNetwork, address destinationAddress, uint256 amount, bytes32 metadataHash) public returns (bytes32) Parameters | Name | Type | Description | | :--- | :--- | :--- | |leafType | uint8 | Leaf type --> [0] transfer Ether / ERC20 tokens, [1] message | |originNetwork | uint32 | Origin Network. | |originAddress | address | [0] Origin token address, 0 address is reserved for ether, [1] msg.sender of the message. | |destinationNetwork | uint32 | Destination network. | |destinationAddress | address | Destination address. | |amount | uint256 | [0] Amount of tokens/ether, [1] Amount of ether. | |metadataHash | bytes32 | Hash of the metadata. | # PolygonAccessControlUpgradeable.md: A contract that inherits AccessControlUpgradeable from Openzeppelin, but with the following modifications: - Deletes ERC165Upgradeable dependencies to save us the "gap" variables and let us have consistent storage. - Adds the legacy Owner variable, to be consistent with the previous. - Adds custom errors. - Replaces msgSender() with msg.sender. Functions AccessControlinit solidity function AccessControlinit( ) internal hasRole Returns true if account has been granted role. solidity function hasRole( ) public returns (bool) checkRole solidity function checkRole( ) internal - Reverts with a standard message if msg.sender is missing role. Overriding this function changes the behavior of the {onlyRole} modifier. - Describes the format of the revert message in {checkRole}. - Available since v4.6. checkRole solidity function checkRole( ) internal - Reverts with a standard message if account is missing role. - The format of the revert reason is given by the following regular expression: /^AccessControl: account (0x[0-9a-f]{40}) is missing role (0x[0-9a-f]{64})$/ getRoleAdmin Returns the admin role that controls role. See grantRole and revokeRole. solidity function getRoleAdmin( ) public returns (bytes32) To change a role's admin, use setRoleAdmin. grantRole Grants role to account. If account has not been granted role, it emits a RoleGranted event. solidity function grantRole( ) public - The caller must have the role as the admin role. - May emit a RoleGranted event. revokeRole Revokes role from account. If account had been granted role, this function emits a RoleRevoked event. solidity function revokeRole( ) public renounceRole Revokes role from the calling account. solidity function renounceRole( ) public - The caller must have the admin role account. - May emit a RoleRevoked event. Roles are often managed via grantRole and revokeRole. The purpose of this function is to provide a mechanism for accounts to lose their privileges if they are compromised (such as when a trusted device is misplaced). Emits a RoleRevokedevent if the calling account had its role revoked. setupRole solidity function setupRole( ) internal Grants role to an account. Emits a RoleGranted event if account has not been granted role before. Note that unlike grantRole, this function doesn't perform any checks on the calling account. May emit a RoleGranted event. !!! warn - This function should only be called from the constructor, when setting up the initial roles for the system. - Using this function in any other way is effectively circumventing the admin system imposed by AccessControl. !!! note - This function is deprecated in favor of grantRole. setRoleAdmin Sets adminRole as the admin role. solidity function setRoleAdmin( ) internal Emits a RoleAdminChanged event. grantRole Grants role to account. Internal function without access restriction. solidity function grantRole( ) internal May emit a RoleGranted event. revokeRole Revokes role from account. Internal function without access restriction. solidity function revokeRole( ) internal May emit a RoleRevoked event. # PolygonRollupBaseEtrog.md: Functions constructor solidity function constructor(contract IPolygonZkEVMGlobalExitRootV2 globalExitRootManager, contract IERC20Upgradeable pol, contract IPolygonZkEVMBridgeV2 bridgeAddress, contract PolygonRollupManager rollupManager) internal Parameters | Name | Type | Description | | :--- | :--- | :--- | |globalExitRootManager | contract IPolygonZkEVMGlobalExitRootV2 | Global exit root manager address. | |pol | contract IERC20Upgradeable | POL token address. | |bridgeAddress | contract IPolygonZkEVMBridgeV2 | Bridge address. | |rollupManager | contract PolygonRollupManager | Global exit root manager address. | initialize solidity function initialize(address admin, address sequencer, uint32 networkID, address gasTokenAddress, string sequencerURL, string networkName) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |admin | address | Admin address. | |sequencer | address | Trusted sequencer address. | |networkID | uint32 | Indicates the network identifier used in the bridge. | |gasTokenAddress | address | Indicates the address of the token used in mainnet as the gas token.

Note that if a wrapped token of the bridge is used, its original network and address are used instead. | |sequencerURL | string | Trusted sequencer URL. | |networkName | string | L2 network name. | sequenceBatches Allows a sequencer to send multiple batches. solidity function sequenceBatches(struct PolygonRollupBaseEtrog.BatchData[] batches, uint64 maxSequenceTimestamp, uint64 initSequencedBatch, address l2Coinbase) public Parameters | Name | Type | Description | | :--- | :--- | :--- | |batches | struct PolygonRollupBaseEtrog.BatchData[] | Struct array which holds the necessary data to append new batches to the sequence. | |maxSequenceTimestamp | uint64 | Max timestamp of the sequence.

The timestamp must be inside a safety range (actual + 36 seconds).

It should be equal or greater than the last block inside the sequence. Otherwise the circuit invalidates the batch. | |initSequencedBatch | uint64 | This parameter must match the current last batch sequenced.

This is a protection mechanism against the sequencer sending undesired data. | |l2Coinbase | address | Address that will receive the fees from L2.

Note that POL is not a reentrant token. | onVerifyBatches It's a callback on verify batches. It can only be called by the rollup manager. solidity function onVerifyBatches(uint64 lastVerifiedBatch, bytes32 newStateRoot, address aggregator) public Parameters | Name | Type | Description | | :--- | :--- | :--- | |lastVerifiedBatch | uint64 | Last verified batch. | |newStateRoot | bytes32 | New state root. | |aggregator | address | Aggregator address. | forceBatch Allows a sequencer/user to force a batch of L2 transactions. This should be used only in extreme cases where the trusted sequencer does not work as expected. solidity function forceBatch(bytes transactions, uint256 polAmount) public !!! note - The sequencer has a degree of control on how non-forced and forced batches are ordered. - In order to assure that users' force transactions are processed properly, each transaction must be signed with a unique nonce. Parameters | Name | Type | Description | | :--- | :--- | :--- | |transactions | bytes | L2 ethereum transactions. EIP-155 or pre-EIP-155 with signature: | |polAmount | uint256 | Max amount of pol tokens that the sender is willing to pay. | sequenceForceBatches Allows anyone to sequence forced batches if the trusted sequencer has not done so within the timeout period. solidity function sequenceForceBatches(struct PolygonRollupBaseEtrog.BatchData[] batches) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |batches | struct PolygonRollupBaseEtrog.BatchData[] | Struct array which holds the data necessary for appending force batches. | setTrustedSequencer Allows the admin to se a new trusted sequencer. solidity function setTrustedSequencer(address newTrustedSequencer) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |newTrustedSequencer | address | Address of the new trusted sequencer. | setTrustedSequencerURL Allows the admin to set the trusted sequencer URL. solidity function setTrustedSequencerURL(string newTrustedSequencerURL) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |newTrustedSequencerURL | string | URL of trusted sequencer. | setForceBatchAddress Allows the admin to change the address allowed to force batches. If address 0 is set, then everyone is able to force batches. This action is irreversible. solidity function setForceBatchAddress(address newForceBatchAddress) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |newForceBatchAddress | address | New force batch address. | setForceBatchTimeout Allows the admin to set the forcedBatchTimeout. The new value can only be lower, except if emergency state is active. solidity function setForceBatchTimeout(uint64 newforceBatchTimeout) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |newforceBatchTimeout | uint64 | New force batch timeout. | transferAdminRole Starts the admin role transfer. This is a two step process. And the pending admin must accept to finalize the process. solidity function transferAdminRole(address newPendingAdmin) external Parameters | Name | Type | Description | | :--- | :--- | :--- | |newPendingAdmin | address | Address of the new pending admin. | acceptAdminRole Allows the current pending admin to accept the admin role. solidity function acceptAdminRole( ) external calculatePolPerForceBatch A function to calculate the reward for a forced batch. solidity function calculatePolPerForceBatch( ) public returns (uint256) generateInitializeTransaction Generates and initializes transaction for the bridge on L2. solidity function generateInitializeTransaction(uint32 networkID, address gasTokenAddress, uint32 gasTokenNetwork, bytes gasTokenMetadata) public returns (bytes) Parameters | Name | Type | Description | | :--- | :--- | :--- | |networkID | uint32 | Indicates the network identifier used in the bridge. | |gasTokenAddress | address | Indicates the token address used to pay gas fees in the new rollup. | |gasTokenNetwork | uint32 | Indicates the native network of the token address. | |gasTokenMetadata | bytes | Abi encoded gas token metadata. | Events SequenceBatches Emitted when the trusted sequencer sends a new batch of transactions. solidity event SequenceBatches( ) ForceBatch Emitted when a batch is forced. solidity event ForceBatch( ) SequenceForceBatches Emitted when forced batches are sequenced by an entity other than the trusted sequencer. solidity event SequenceForceBatches( ) InitialSequenceBatches Emitted when the contract is initialized. It contains the first sequenced transaction. solidity event InitialSequenceBatches( ) VerifyBatches Emitted when an aggregator verifies batches. solidity event VerifyBatches( ) SetTrustedSequencer Emitted when the admin updates the trusted sequencer address. solidity event SetTrustedSequencer( ) SetTrustedSequencerURL Emitted when the admin updates the sequencer URL. solidity event SetTrustedSequencerURL( ) SetForceBatchTimeout Emitted when the admin updates the force batch timeout. solidity event SetForceBatchTimeout( ) SetForceBatchAddress Emitted when the admin updates the force batch address. solidity event SetForceBatchAddress( ) TransferAdminRole Emitted when the admin starts the two-step transfer role of setting a new pending admin. solidity event TransferAdminRole( ) AcceptAdminRole Emitted when the pending admin accepts the admin role. solidity event AcceptAdminRole( ) # PolygonTransparentProxy.md: Inherits from TransparentUpgradeableProxy, a Openzeppelin v5 contract, with the following modifications: - Admin is a parameter in the constructor instead of being deployed. - Let the admin get access to the proxy. - Replace msgSender() with msg.sender Functions constructor Initializes an upgradeable proxy managed by an instance of a ProxyAdmin with an initialOwner backed by the implementation at logic, and optionally initialized with data as explained in ERC1967Proxy-constructor. solidity function constructor( ) public proxyAdmin Returns the admin of this proxy. solidity function proxyAdmin( ) internal returns (address) fallback If caller is the admin, process the call internally. Otherwise, transparently fallback to the proxy behavior. solidity function fallback( ) internal # PolygonZkEVMGlobalExitRootBaseStorage.md: Abstract contract required to preserve the storage slots from previous versions. Inherits from IPolygonZkEVMGlobalExitRootV2.sol Functions None. Variables lastRollupExitRoot Rollup root, contains all exit roots of all rollups. Returns bytes32 object. lastMainnetExitRoot Mainnet exit root; updated every time a deposit is made in mainnet. Returns bytes32 object. globalExitRootMap Stores every global exit root: where root == blockhash. Note that global exit roots in previous versions recorded timestamp instead of blockhash. Returns mapping(bytes32 => uint256) object. # ClaimCompressor.md: Utility contract for compressing and decompressing claim data. Functions constructor solidity function constructor(address bridgeAddress, uint32 networkID) public Parameters | Name | Type | Description | | :--- | :--- | :--- | |bridgeAddress | address | PolygonZkEVMBridge contract address. | |networkID | uint32 | Network ID. | compressClaimCall Forwards all claim parameters in order to compress them inside the contract. solidity function compressClaimCall(bytes32 mainnetExitRoot, bytes32 rollupExitRoot, struct ClaimCompressor.CompressClaimCallData[] compressClaimCalldata) external returns (bytes) Parameters | Name | Type | Description | | :--- | :--- | :--- | |mainnetExitRoot | bytes32 | Mainnet exit root. | |rollupExitRoot | bytes32 | Rollup exit root. | |compressClaimCalldata | struct ClaimCompressor.CompressClaimCallData[] | compress claim calldata. | sendCompressedClaims solidity function sendCompressedClaims( ) external # admin-role.md: Polygon zkEVM's Admin consists of three (3) developers of the Polygon team, who oversee any upgrades of the zk-rollup software. Whenever there are bug fixes or updates to be made, the Admin uses a special Admin Multisig Contract to approve the changes. Approval by any two of the three Admin members (2-out-of-3) is required. However, there's a minimum 10-day waiting period before the changes get executed. The 10-day delay allows users to carefully assess the proposed changes and decide whether to exit or not. Another smart contract called Timelock Contract is responsible for enabling the 10-day delay. An outline of the upgrade process is outlined here. Admin contract in detail The Admin owns the Ethereum account that controls the Consensus contract, and it is the only account that can call the following functions; - setTrustedSequencer - setForceBatchAllowed - setTrustedSequencerURL - setTrustedAggregator - setTrustedAggregatorTimeout - setPendingStateTimeout - setMultiplierBatchFee - setVeryBatchTimeTarget - setAdmin - deactivateEmergencyState All ProxyAdmin.sol instances that can upgrade the contract implementations of the zkEVM Protocol belong to the Admin account. Moreover, all proxies are owned by the Admin account, making it the only account (other than the Security Council Multisig) authorized to make modifications to the contracts used to implement the zkEVM Protocol. Timelock controller !!!tip What is Timelock Controller Timelock Controller is a smart contract that enables setting up a delay to provide users some time to leave before applying potentially risky maintenance procedures. The Timelock Controller has been added to the zkEVM Protocol in order to improve user security and confidence. The Admin can schedule and commit maintenance operations transactions in L1 using a Timelock Controller, and the timelock can be activated to carry out the transactions when a specified minDelay time has passed. The Polygon zkEVM team has decided to use the OpenZeppelin's TimelockController.sol contract to inherit security as well to avoid the lengthy and complicated audit process. We have changed the getMinDelay method in the contract and this modified implementation is named PolygonZkEVMTimelock.sol contract. In the event that the zkEVM contract system is in Emergency Mode, the new getMinDelay sets the time minDelay to 0. This is when the Security Council Multisig takes control. The zkEVM Protocolâ€™s Admin role is set to an instance of PolygonZkEVMTimelock.sol contract address since the deployment of the zk-rollup. Governance of zkEVM contracts The Admin carries significant and critical responsibility, which is why it is composed of three (3) members as opposed to just one person. For this reason, the Admin Ethereum account of a PolygonZkEVMTimelock.sol contract instance is assigned to a multisig contract that acts as a governance tool for the zkEVM Protocol, therefore achieving some degree of overall decentralization. Below figure shows the governance tree of Polygon zkEVM L1 contracts. !governance tree of zkEVM L1 contracts Protocol maintenance operations can only be performed by following these steps: !!!info Due to the governance chain among protocol contracts, all transactions generally follow the below steps. 1. Maintenance operations transactions are proposed and stored into governance multisig contract. Polygon team reaches a consensus on whether or not to apply these operations. Voting inherits security from L1. 2. Once a decision has been reached, and the results are favorable to perform maintenance operations, the governance multisig can be triggered to schedule the transactions to be executed but only when the 10-day time delay has passed. 3. Once a time delay has passed, the Timelock Contract dubbed PolygonZkEVMTimelock.sol executes the scheduled transactions. # etrog-upgrade.md: Etrog upgrade This document provides details of the Etrog upgrade, which is Polygon zkEVM's upgrade that succeeds the Dragonfruit upgrade. Although the Dragonfruit upgrade had some advantages over previous zkEVM versions, it has its own pain points. We take a look at what these pain points are, and how the Etrog upgrade resolves them. Before discussing all these, we present the newly launched testnet that coincides with the Etrog upgrade. Sepolia public testnet: Cardona In addition to the GoÃ«rli testnet, we launched a second testnet on Sepolia dubbed Cardona. Due to the looming deprecation of the GoÃ«rli testnet, the Polygon team had to put in place a second testnet. This ensures that the zkEVM can continue running even beyond the GoÃ«rli's discontinuation. The Cardona testnet is ready for developers and users to connect wallets, and begin testing the Etrog upgrade of Polygon zkEVM. Use the following details to connect wallets to Cardona: - Network Name: Polygon zkEVM Cardona Testnet - Bridge UI: https://bridge-ui.cardona.zkevm-rpc.com - New RPC URL: https://rpc.cardona.zkevm-rpc.com - Chain ID: 2442 - Currency symbol: ETH - Explorer: https://cardona-zkevm.polygonscan.com/ We use the same faucet for testnet tokens, here: https://faucet.polygon.technology/ zkEVM is almost type 2 The Etrog upgrade comes with support for most of the EVM's precompiled contracts; ecRecover, SHA-256, identity, modexp, ecAdd, ecMul, and ecPairing. This only leaves out the barely used RIPEMD-160 and blake2f. Owing to this fact, the Etrog upgrade therefore helps Polygon zkEVM inch closer to becoming a type 2 ZK-EVM. The below table displays Polygon zkEVM's precompiled status. !Figure: etrog-precompiled Supported opcodes | OPCODE | Support status | Details | | ---------------- | -------------- | -- | | SELFDESTRUCT | Not supported | Replaced by SENDALL. | | EXTCODEHASH | Supported | It returns the hash of the contract bytecode from the zkEVM state tree without checking if the account is empty. | | DIFFICULTY | Supported | It returns "0" instead of a random number as in the EVM. | | BLOCKHASH | Supported | It is the state root at the end of a processable transaction and is stored on the system smart contract. It returns all previous block hashes instead of just the last 256 blocks. | | NUMBER | Supported | It returns the number of processable transactions. | | JUMPDEST | Supported | It is allowed in PUSH bytes to avoid runtime bytecode analysis. | | BASEFEE | Not supported | The zkEVM implements the Berlin hardfork, but not the London hardfork. | EIPs support | EIPs | Hardfork | Support status | Description | | -------- | -------- | ------------- | --- | | EIP-2718 | Berlin | Not supported. | Defines a new transaction type that is an envelope for future transaction types.[^e1] | | EIP-2930 | Berlin | Not supported. | Defines the Optional Access Lists transaction type.[^e2] | | EIP-3541 | London | Supported. | Reject new contract code starting with the 0xEF byte.[^e3] | Additions zk-counters indicate batch resources that are available, linked to state-machine components, as a supplementary addition to gas computation. Dragonfruit issues Dragonfruit upgrade inherited from its predecessors the configuration that each block in the zkEVM is equivalent to one L2 transaction. The justification for the one block per transaction configuration was that it achieves minimum delay when creating blocks. It however results in a few issues when blocks are processed. The first issue is that it generates a lot of data in the database due to the huge amount of L2 blocks being created. The second being, the approach lacks a way to provide different timestamps for blocks within a batch. So, all blocks in a batch have the same timestamp. However, attaching one timestamp to several blocks causes breaks in dApps that rely on timestamps for proper timing of smart contracts' actions. The crucial alterations made in the Etrog upgrade are therefore on; the one block per transaction approach, and the one timestamp for many blocks problem. Etrog blocks Up until the Dragonfruit upgrade, each L2 block contained a single transaction. This is the same approach taken by Optimism. Since a block is formed when the sequencer decides to include a transaction in a batch, prior to the Etrog upgrade, every batch contained as many blocks as transactions. As mentioned above, this resulted in bloated databases. Due to this problem, blocks in the Etrog upgrade have been reconstructed to contain more than one transaction. Blocks with several transactions are achieved by using a small timeout of a few seconds, or a number of milliseconds, when creating the block and waiting for transactions to be added. The figure below shows the structure of the blocks in Etrog versus those of the Dragonfruit upgrade. !Figure: etrog-blocks-vs-dragonfruit Etrog timestamps In order to circumvent the above-mentioned issue related to the one timestamp for many blocks problem, each block in the Etrog upgrade's batch receives its own unique timestamp. This is in addition to allowing more than one transaction per block. The solution is achieved by enabling the sequencer to change the timestamp for different blocks within a batch. To do so, a special transaction or marker, called changeL2Block, is introduced within a batch to mark whenever there is a block change. The figure below shows how changeL2Block is used to change the timestamp whenever a new block is formed. !Figure: etrog-changeL2block Conclusion The Etrog upgrade comes with groundbreaking amendments aimed at improving UX and developer experience. The most important additions with this upgrade are the two adjustments that solve the two issues mentioned-above: - Being able to add multiple transactions to one block. - Allowing granularity on the timestamp within a batch. Attaining the Type-2 status is remarkable. [^e1]: https://eips.ethereum.org/EIPS/eip-2718 [^e2]: https://eips.ethereum.org/EIPS/eip-2930 [^e3]: https://eips.ethereum.org/EIPS/eip-3541 # incentive-mechanism.md: In order to ensure the system's sustainability, actors must be compensated for correctly performing their roles and giving the protocol finality. Unless otherwise specified, the measures and rules presented here apply to cases in which the Sequencer and Aggregator roles are decentralised (i.e., when there are no trusted sequencer and no trusted aggregator). L2 transaction fees and sequencing fees The native currency used in L2 is Bridged Ether, which originates from L1. This is the currency that is used to pay L2 transaction fees. It can be transferred at a 1:1 exchange ratio from L1 to L2 and vice versa. The Sequencer earns the transaction fees paid by L2 users for submitting transactions, and thus gets paid directly in Bridged Ether. The amount of fees paid depends on the gas price, which is set by users based on how much they are willing to pay for the execution of their transactions. To incentivize the Aggregator for each batch sequenced, the Sequencer must lock a number of POL tokens in the L1 PolygonZkEVM.sol Contract proportional to the number of batches in the sequence. The number of POL tokens locked per batch sequenced is saved in the variable batchFee. The below diagram depicts the various fees and rewards earned by the protocol's actors. !Fees paid and rewards rewards for each actor in the protocol To maximize its income, the Sequencer prioritizes transactions with higher gas prices. Furthermore, there is a threshold below which it is unprofitable for the Sequencer to execute transactions because the fees earned from L2 users are less than the fees paid for sequencing fees (plus L1 sequencing transaction fee). Users must ensure that their transaction fees are greater than this threshold in order for the Sequencer to be incentivized to process their transactions. The net Ether value earned by the Sequencer for sequencing a batch sequence is represented by the following expression: $$ \textit{ }\mathtt{Sequencer\ net\ Ether\ income}\textit{" } \mathtt{ =\ totalL2TxGasFee â^' (L1SeqTxGasFees + \frac{batchFee â— nBatches}{POL/ETH})} $$ where: - totalL2TxGasFees is the total sum of fees gathered from all L2 transactions included in the sequence of batches, - L1SeqTxGasFee is the Sequencing transaction gas fee paid in L1, - batchFee is the storage variable in PolygonZkEVM.sol contract, - nBatches is the number of batches in the sequence, - POL/ETH is the price of POL token expressed in ETH. Aggregation reward The Aggregator also needs compensation for correctly fulfilling its role. The number of POL tokens earned by the Aggregator each time it aggregates a sequence, denoted by batchReward, is determined by the total contract POL balance and the number of batches aggregated. The POL earned per batch aggregated is calculated by the L1 PolygonZkEVM.sol contract prior to sequence aggregation using the following expression: $$ \mathtt{batchReward} = { \dfrac{ \textit{}contract\ POL\ balance\textit{"}}{\textit{}Quantity\ of\ batches\ not\ aggregated\ yet \textit{"}}} $$ The following expression represents the total amount of ETH value that the Aggregator earns for the aggregation of a sequence of batches: $$ \mathtt{\textit{ }Aggregator net Ether income\textit{"}= \frac{batchReward â— nBatches}{POL/ETH} â^' L1AggTxGasFee} $$ where: - L1AggTxGasFee is the Aggregation transaction gas fee paid in L1, - batchReward is the quantity of POL earned per batch aggregated, - nBatches is the number of batches in the sequence, - POL/ETH is the price of POL token expressed in ETH. Variable batchFee re-adjustments The batchFee is automatically adjusted with every aggregation of a sequence by an independent Aggregator. This happens when the trusted aggregator isn't working properly and the batchFee variable needs to be changed to encourage aggregation. Further information on the trusted aggregator's inactivity or malfunctioning is provided in upcoming sections. An internal method called updateBatchFee, is used to adjust batchFee storage variable. pil function updateBatchFee(uint64 newLastVerifiedBatch) internal The admin defines two storage variables that are used to tune the fee adjustment function: - veryBatchTimeTarget, which is the targeted time of the verification of a batch, so the batchFee variable is updated to achieve this target, and - multiplierBatchFee, which is the batch fee multiplier, with 3 decimals ranging from 1000 to 1024. The function updateBatchFee first determines how many of the aggregated batches are late. That is, those who are in the sequence but have not yet been aggregated. Second, how much time has passed, as indicated by veryBatchTimeTarget. The diffBatches variable represents the difference between late batches and those below the target, and its value is limited by a constant called MAX BATCH MULTIPLIER, which is set to 12. Case 1 If there are more late batches than early batches in the sequence being aggregated, the following formula is applied to the batchFee storage variable: $$ \textit{ }\mathtt{ new\ batch\ fee}\textit{â€ } \ \mathtt{ =\ } \textit{ } \mathtt{ old\ batch\ fee \textit{â€ } \cdot \big(\frac{multiplierBatchFee^{diffBatches}}{10^{3\ diff Batches}} \big) } $$ The graph below shows the percentage variation of the 'batchFee' variable depending on the 'diffBatches' value for different values of'multiplierBatchFee' when late batches dominate the sequence. It should be noted that the goal is to increase the aggregation reward in order to incentivize aggregators. !% of batch fee variation when late batches dominate the sequence Case 2 If the early batches outnumber the late ones, the following formula is applied to the storage variable 'batchFee': $$ \mathtt{ "new\ batch\ fee" = "old\ batch\ fee" \cdot \big(\frac{10^{3\ diffBatches}}{multiplierBatchFee^{diffBatches}} \big) } $$ The graph below shows the percentage variation of the batchFee variable depending on the diffBatches value for different values of multiplierBatchFee when batches below the time target dominate the sequence. It should be noted that the goal is to reduce the aggregation reward. !% of batch fee variation when batches below the time target dominate the sequence To summarize, the admin can tune the reaction of batchFee variable re-adjustments by adjusting veryBatchTimeTarget and multiplierBatchFee. The values set during the contract's initialization are listed below: - batchFee = 1018 (1 POL). - veryBatchTimeTarget = 30 minutes. - multiplierBatchFee = 1002. # index.md: Over the past few years, several L2 solutions have been developed to enhance the scalability of the Ethereum network, with a focus on increasing transaction throughput. The benefit for users of the Ethereum network is to lower gas fees without compromising decentralization and security. Polygon zkEVM is an L2 rollup solution that combines data availability and execution verification on L1, the Ethereum network, in order to ensure security and reliability of each L2 state transition. This section describes the overall design of the Polygon zkEVM. It thus provides an architectural overview of its protocol. Components It takes three main components of the Polygon zkEVM protocol to enable transaction finality while ensuring the correctness of state transitions: - The trusted sequencer. - The trusted aggregator. - The consensus contract (PolygonZkEVM.sol, deployed on L1). Trusted sequencer The trusted sequencer receives L2 transactions from users, orders them, generates blocks of transactions, fills batches, and submits them to the consensus contract's storage slots in the form of sequences. The trusted sequencer processes batches and distributes them to L2 network nodes to achieve immediate finality and reduce costs associated with high network usage, all before submitting them to L1. The trusted sequencer runs a zkEVM node in sequencer mode and controls an Ethereum account regulated by a consensus contract. Trusted aggregator The trusted aggregator computes the L2 state based on batches of L2 transactions executed by the trusted sequencer. On the other hand, the primary function of the trusted aggregator is to receive the L2 batches validated by the trusted sequencer and produce zero-knowledge proofs verifying the computational integrity of these batches. The aggregator achieves this by employing a specialized off-chain EVM interpreter to generate the ZK proofs. The logic within the consensus contract verifies the zero-knowledge proofs, thereby endowing the zkEVM with the security of Layer 1. Before committing new L2 state roots to the consensus contract, verification is essential. A validated proof serves as undeniable evidence that a particular sequence of batches resulted in a specific L2 state. !!!info L2 state root An L2 state root is a cryptographic hash value of the L2 state. In case you want to read more about state roots, please check out this article. The trusted aggregator runs a zkEVM node in aggregator mode and controls an Ethereum account regulated by a consensus contract. Consensus contract The consensus contract used by both the trusted sequencer and the trusted aggregator in their interactions with L1 is the PolygonZkEVM.sol contract. The trusted sequencer can commit batch sequences to L1 and store them in the PolygonZkEVM.sol contract, creating a historical repository of sequences. The PolygonZkEVM.sol contract also enables the aggregator to publicly verify transitions from one L2 state root to the next. The consensus contract accomplishes this by validating the aggregator's zk-proofs, which attest to proper execution of transaction batches. zkEVM node execution modes zkEVM node is a software package containing all components needed to run zkEVM network. It can be run in three different modes; as a sequencer, an aggregator, or RPC. Sequencer mode In the sequencer mode, the node holds an instance of L2 state, manages batch broadcasting to other L2 network nodes, and has a built-in API to handle L2 user interactions (transaction requests and L2 State queries). There is also a database to temporarily store transactions that have not yet been ordered and executed (pending transactions pool), as well as all the components required to interact with L1 in order to sequence transaction batches and keep its local L2 state up to date. Aggregator mode In the aggregator mode, the node has all the components needed to execute transaction batches, compute the resulting L2 state and generate the zero-knowledge proofs of computational integrity. Also, it has all the components needed to fetch transaction batches committed in L1 by the trusted sequencer and call the functions to publicly verify the L2 state transitions on L1. RPC mode In this mode, the zkEVM node has limited functionality. It primarily maintains an up-to-date instance of the L2 state, initially with respect to batches broadcast by the trusted sequencer, and later with sequences of batches fetched from the consensus contract. The RPC node continuously interacts with L1 to keep the local L2 state up-to-date and to verify the synchronization of L2 state roots. By default, the synchronizer updates every 2 seconds, unless a different interval is specified in the configuration. # security-council.md: In addition to the previously mentioned governance issues and security measures, one more component was essential, especially to a young zk-rollup such as the Polygon zkEVM. That component is, the Security Council Multisig. Since critical bugs or other security issues may occur, and hence warrant instant upgrades, it is good security practice to allow for emergency upgrades. That is, instead of employing the 2-out-of-3 Admin Multisig Contract and waiting for the time-delay imposed by the Timelock Contract, the Security Council Multisig may actvate the emergency state to bypass such time-delay. It is crucial, however, to emphasise that the Security Council Multisig is a temporary measure, and will ultimately be phased-out once the Polygon zkEVM has been sufficiently battle-tested. Understanding security council multisig The security council is a committee that oversees the security of the Polygon zkEVM during its initial phase. The security council of a rollup has a two-fold responsibility, - Seeing to it that the system is timeously halted in case of the emergency state, and - Ensuring that emergency upgrades are implemented as soon as it is practically possible. The security council therefore utilises a special multisig contract that overrides the usual Admin Multisig Contract and the Timelock Contract. !Figure 1: Overview of the Security Council in relation to the Admin Contract Security council composition Security councils generally consist of a certain number of reputable community members, who are typically, individuals or representatives of public organizations who may remain anonymous. These are individuals or organizations with vested interest in the welfare of the Ethereum ecosystem, and are normally selected from among well-known Ethereum developers and researchers. The Polygon zkEVM's Security Council is constituted of eight (8) members, two (2) of whom are internal to the Polygon team, while the rest of the members must be from outside Polygon. The minimum requirement, even as mentioned in the L2Beat report downloadable here, is for these individuals to be adequately knowledgeable and competent enough to make the best judgment about the actions approved by the multisig. These members are not completely anonymous as their addresses are publicly known. Their addresses can be checked in Etherscan. Here is a list of the 8 addresses of the Polygon zkEVM's Security Council; - 0xFe45â€¦2e4b - 0xaF46â€¦261D - 0xBDc2â€¦FEFf - 0x4c16â€¦8891 - 0x3ab9â€¦D622 - 0x49c1â€¦0E86 - 0x9F7dâ€¦86A0 - 0x2188â€¦1C28 Security council multisig? The Security Council Multisig is a multisig contract deployed by the Polygon zkEVM Security Council that allows the emergency state to be triggered, in the case an emergency upgrade needs to be executed. The multisig contract is a 6-out-of-8 multisig, which requires six (6) signatures of the Security Council to be attached for the emergency state to be triggered. There is a further stipulation that a minimum of 4 out of the 6 attached signatures must be from among the 6 members who are external to Polygon. Conclusion Although the ultimate goal is to move towards a totally decentralized Polygon zkEVM, employing a security council multisig is inevitable for the early stages of the zkRollup. It is a trade-off between security and decentralization. So then, for the sake of long-term security, it is a deliberate decision to have more centralized early stages of development, in order to attain more decentralized later stages. Although there is always a possibility for the members of Security Council to go rogue and collude, the 75% threshold together with the minimum 66% of external membersâ€™ signatures significantly reduces the risk. # sequencing-batches.md: The central entity responsible for assembling batches for sequencing is the trusted sequencer, which is built and managed by Polygon. This sequencer may omit Layer 2 transactions therefore we have implemented anti-censorship mechanism. The diagram below shows the sequencing workflow. !Figure: 1. L2 transactions via JSON RPC. 2. Transactions stored in the pool DB. 3. Trusted sequencer selects transactions from pool DB. 4. Trusted sequencer batches and sequences transactions. Batch pre-execution The initial step in creating a batch involves verifying whether the chosen transactions align with execution parameters and do not surpass the gas limit. This step is known as batch pre-execution. It is carried out by the sequencer through an executor, as depicted in the figure below. !Figure: Pre-execution While no proof is generated during the pre-execution stage, batch pre-execution ensures that the prover's subsequent proof generation is successful, and expedites batch sequencing overall. A fast executor is used for batch pre-execution. This is a single-computation executor, capable of executing within blocktime. The sequencer communicates with the executor for swift batch pre-execution. Once the executor successfully completes batch pre-execution, the sequencer records the batch in the nodeâ€™s StateDB as a closed batch. A closed batch means one of the following conditions has been fulfilled: - The execution trace reaches the maximum number of rows. - The gas used attains maximum gas limit. - The allocated time expires. During batch pre-execution, and for batch closure, the sequencer and the executor update the Merkle tree of the zkEVM with L2 state changes and store them in the prover's hash DB. This is illustrated in the figure below: !Figure: Update L2 state The zkEVM's throughput depends on the speed at which we are able to close batches, which is directly impacted by the batch pre-execution process. Performance problems can occur here because of excessive and inefficient interaction with the hash DB. Optimizing this process may mean reducing the time spent on frequent updates during transaction processing by accumulating all state changes caused by a transaction and only updating the database at the end of a transaction's pre-execution. Sending batches to L1 The next step is to send a call to the smart contract to sequence batches. Once a batch is closed, the sequencer stores the data of the batch in the nodeâ€™s state DB. Then, the $\texttt{sequenceSender}$ looks for closed batches and sends them to the L1 smart contract via the $\texttt{EthTxManager}$ which makes sure the transaction is included in a batch. This process is depicted in the figure below. !Figure: Sequence sender and ETH Tx Manager In order to sequence a batch, the sequencer calls the $\texttt{sequenceBatches()}$ function in the L1 smart contract which can sequence one or multiple batches at once. The calldata for the L1 $\texttt{sequenceBatches()}$â€‹ function needs to include the following information: - The L2 transactions data, which is an array containing data for each batch. It includes all transactions within the batch along with a timestamp indicating its closure time. - Additionally, the L2 coinbase address, representing the Ethereum address for receiving user fees. - Lastly, a timestamp indicating when the L2 transactions were sequenced. The L2 coinbase address serves as a critical destination for rewards earned by the sequencer in the Layer 2 environment. The sequencer is responsible for paying for data availability in layer 1 using L1 Ether. The sequencer receives a reward for successfully closing a batch and executing transactions. This reward, denominated in L2 Ether, is routed to the L2 coinbase address. The L2 coinbase address is situated within layer 2 because users compensate the sequencer with L2 Ether. This L2 Ether, representing the reward, is a mapping of the Ether in L1 that users have previously transferred to L2 through transactions via the bridge. There is a direct and fixed one-to-one correspondence between L1 ETH and L2 ETH, as we can observe in the figure below. !Figure: L1 ETH and L2 ETH equivalence Accumulated input hash pointers When the smart contract receives a transaction for sequencing into batches, it creates a cryptographic pointer for each batch. These pointers identify a batch and specify its position. Subsequently, provers utilize these pointers as references during the proving process, allowing them to precisely identify the batch being proved and retrieve its associated data. The use of cryptographic pointers ensures a robust and unambiguous link between the sequencing operation and the corresponding batch data for subsequent verification. !Figure: Sequence of batches - ... timeline The pointers are constructed with previous hashes and information from the batches. The procedural steps for this process are illustrated in the figure below: !Figure: Stringing together batch hash data Pointers are generated by executing the KECCAK hash function on: - The preceding pointer. - The transactions encompassed within the L2 batch. - The batch timestamp. - The L2 coinbase. A pointer is referred to as an accumulated input hash or $\texttt{accInputHash}$. Such a construction, where previous pointers are linked to the current pointer, guarantees that batch data is proved in the correct and sequential order. When the sequencing process completes, the batch enters a virtual state. # state-management.md: This document explains how the Polygon zkEVM protocol manages the L2 rollup state while providing verifiability of each state transition. Trustless L2 state management The trusted sequencer generates batches and broadcasts the batches to all L2 nodes. Each L2 node can then run the batches to compute the resulting L2 state locally, and thus achieve faster finality of L2 transactions. But the trusted sequencer also commits the batches to L1, allowing L2 nodes who rely on the Ethereum security to fetch the batches from there, execute them, and thus compute the L2 state. Nodes requiring stringent security can wait for correct transactional computations to be proved and verified, before syncing state. Zero-knowledge proofs are computed off-chain as required by the L1 smart contract and proofs are verified by another verifier smart contract. This way both data availability and verification of transaction execution rely only on L1 security assumptions. !figure 1 As shown in the above figure, L2 nodes can receive batch data in three different ways: 1. Directly from the trusted sequencer before the batches are committed to L1. 2. Straight from L1 after the batches have been sequenced. 3. Only after correctness of execution has been proved by the aggregator and verified by the PolygonZkEVM.sol contract. It is worth noting that the three batch data formats are received by L2 nodes in the chronological order listed above. Three L2 states There are three stages of the L2 state, each corresponding to the three different ways in which L2 nodes can update their state. All three cases depend on the format of batch data used to update the L2 state. In the first instance, the update is informed solely by the information (i.e., Batches consisting of ordered transactions) coming directly from the trusted sequencer, before any data availability on L1. The resulting L2 state is called the trusted state. In the second case, the update is based on information retrieved from the L1 network by L2 nodes. That is, after the batches have been sequenced and data has been made available on L1. The L2 state is referred to as the virtual state at this point. The information used to update the L2 state in the last case includes verified zero-knowledge proofs of computational integrity. That is, after the zero-knowledge proof has been successfully verified in L1, L2 nodes synchronize their local L2 state root with the one committed in L1 by the trusted aggregator. As a result, such an L2 state is known as the consolidated state. The figure below depicts the timeline of L2 State stages from a batch perspective, as well as the actions that trigger progression from one stage to the next. !L2 State stages timeline # synchronizer-reorg.md: This document explores how Polygon zkEVM deals with reorganizations, or simply reorgs, compared to Ethereum reorgs, and discusses the role the synchronizer plays in these reorgs. Among other tasks, the synchronizer is responsible for managing the reorg process. L2 reorgs Consider a reorg of L2 batches. Suppose a sequencer, called sequencer A, has closed batch $\mathtt{724}$ and has not sequenced it yet. So the batch, denoted by $\mathtt{724^A}$, is part of the trusted state. And thus, the figure below depicts batch $\mathtt{724^A}$ in red. !Figure: Sequencer and 3 states However, suppose that another sequencer, called sequencer B, closes and sequences a different batch $\mathtt{724}$. The batch, denoted by $\mathtt{724^B}$, is therefore part of the virtual state. The figure below depicts batch $\mathtt{724^B}$ in green. !Figure: Reorg - consolidated state Therefore, to align with the current virtual state, sequencer A must re-synchronize its state from batch $\mathtt{724^B}$ onwards. To accomplish this, sequencers must always check sequenced transactions present in the L1, in case another sequencer has virtualized a different batch. In the zkEVM architecture, a component called the synchronizer is responsible for checking events emitted in L1 when batches are sequenced. This way the sequencer can re-synchronize if necessary. !Figure: Sequencer resync In general, the synchronizer checks Layer 1 for instances of sequenced batches. If two or more sequencers, $\mathtt{A}$, $\mathtt{B}$, $\mathtt{C}$, $\mathtt{ \dots }$, have closed different $\mathtt{X}$-th batches, $$ \mathtt{batch\ X^Ai},\ \mathtt{batch\ X^Bi},\ \mathtt{batch\ X^Ci},\ \dots $$ and some of these $\mathtt{X}$-th batches have been virtualized and some consolidated, then the synchronizer must notify, - The sequencers whose batches are 'un-virtualized', and - The sequencers whose batches are 'un-consolidated', of the need to reorg. And, as in the above scenario of sequencer A and B, the synchronizer alerts sequencer A of the need to reorg. !!! info Currently, reorgs do not occur in the zkEVM system because a single sequencer is implemented, the trusted sequencer. The possibility of L2 reorgs arises only in the event of L1 reorgs. Thereâ€™s a slight chance for a reorg occurrence in the event of forced batches, where a user is sequencing a batch. However, the design of forced batches is specifically crafted to mitigate such scenarios. L1 reorgs L1 reorgs happen if there is a reorg in Ethereum itself. In general, L1 reorgs should never happen because once a state is consolidated, then users have the guarantee that the transactions are finalized. L1 reorgs are far more critical, since it might be the case that the sequencer has to re-synchronize already virtualized or consolidated batches. A reorg in L1 requires a change of the state. The figure below, depicts a reorg scenario where two batches are in the virtual state and one batch is in the consolidated state. !Figure: Reorg in L1 requires state change The synchronizer is responsible for identifying such scenarios and informing the sequencer to perform the appropriate reorg. Synchronizer Although the synchronizer is essential for performing reorgs, it is generally needed for detecting and recording any relevant event from L1 (not just reorgs) in the nodeâ€™s StateDB. # upgradability.md: It is inevitable for the current version of the Polygon zkEVM to go through updates and some upgrades, as it gets tested by both the community of Polygon developers and the internal team. For this reason, and as an effort towards incentivising developers to battle-test the Polygon zkEVM, bug-bounties have been made available. Since zk-Rollup ecosystems are nascent, it is expected that the frequency of upgrades should decline with time. In tandem, Polygon intends to move its governance of upgrades from the currently centralized approach to a much more decentralized modus operandi. These gradual changes in governance follow Polygon Improvement Proposals (PIPs), as already outlined in Polygon's Three Pillars of Governance. Presently, centralization is seen in the form of the Admin Multisig Contract and the Security Council Multisig. Deploying battle-tested contracts To allow for future updates to the zkEVM Protocol implementation (either in the case of adding new features, fixing bugs, or optimizations upgrades), the following contracts are deployed using a Transparent Upgradeable Proxy (TUP) pattern: - PolygonZkEVM.sol (Consensus Contract) - PolygonZkEVMGlobalExitRoot.sol - PolygonZkEVMBridge.sol To inherit security and avoid prolonging and making the audit process more complex, the Polygon zkEVM team has chosen to use the OpenZeppelinâ€™s openzeppelin-upgrades library in order to implement this functionality. !!!info Why Use OpenZeppelin Libraries? OpenZeppelin is a reputated and well-known brand in the industry because of its audits and open-source libraries of implementations of Ethereum standards, and its openzeppelin-upgrades library has been already audited and battle tested. Furthermore, openzeppelin-upgrades is more than just a set of contracts; it also includes Hardhat and Truffle plugins to help with proxy deployment, upgrades, and administrator rights management. As shown in the diagram below, Open Zeppelin's TUP pattern separates the protocol implementation of storage variables using delegated calls and the fallback function, allowing the implementation code to be updated without changing the storage state or the contract's public address. !tup pattern schema Following OpenZeppelin's recommendations, an instance of the contract ProxyAdmin.sol, which is also included in the openzeppelin-upgrades library, is deployed and its address is set as the proxy contract's admin. The Hardhat and Truffle plugins make these operations safe and simple. Each ProxyAdmin.sol instance serves as the actual administrative interface for each proxy, and the administrative account is the owner of each ProxyAdmin.sol instance. The ownership of ProxyAdmin.sol was transferred to the Admin role when the zkEVM Protocol was launched. # upgrade-process.md: For the sake of securing the Polygon zkEVM, which is still in its Beta version, it is best to catch and prevent any possible vulnerabilities now than later. Although Upgradeability is not a permanent feature of the Polygon zkEVM but only a part of the so-called Training Wheels, this document acts as a note on the process followed when upgrading. Upgrades on the Polygon zkEVM typically affect the following contracts: - PolygonZkEVM.sol (Consensus Contract) - PolygonZkEVMGlobalExitRoot.sol - PolygonZkEVMBridge.sol (Bridge Contract) A typical upgrade can only change the logic but not the state of the network. For instance, an upgrade affecting the zkEVM's Consensus Contract (or PolygonZkEVM.sol) could be changing the old verifier contract to a new one. In this case, the logic changes from pointing to the old verifier contract to the new one, leaving the state intact. Security parameters The security measures taken by the zkEVM team for an upgrade are on par with Ethereum's security standards as they involve the deployment of; - An Admin Multisig Contract to avoid having one account controlling upgrades, - A Timelock Contract to give users sufficient time delay to withdraw before execution, and - A Transparent Upgradeable Proxy, from OpenZeppelin's libraries of audited and battle-tested contracts. Process overview As the need arises, and while Upgradeability is still permissible, a proposal for an upgrade can be made. Before being sent to the Timelock Contract, the proposal needs to be signed by 2 out of 3 eligible signatories via the Admin Multisig Contract. Once the conditions of the Admin Multisig Contract are satisfied, including a minimum of two signatures having been attached, scheduling of the proposed upgrade can be made with the Timelock Contract. The time delay set for a zkEVM's upgrade is 10 days, after which the Admin triggers the Timelock Contract to execute the upgrade. This means the upgrade gets submitted to the L1 as a normal transaction. In line with Transparent Upgradaeable Proxies, this ensures that the state of the zkEVM remains intact while the logic gets changed. Following the above example of an upgrade on the Consensus Contract, the below depicts the process flow of a Polygon zkEVM upgrade. !Upgrade Overview Benefits for users Firstly, the zkEVM team is committed to securing the system for the sake of protecting users' funds. As a result, any perceived threat to security, whether big or small, needs to be nipped in the bud. Secondly, most upgrades often include optimizations, bug fixing, or a more accurate formula for effective gas pricing. This subsequently means fair and less transaction costs overall. Polygon's governance position Polygon is committed to aligning itself with Ethereum's norms and values regarding L2 Governance. For more details on Polygon's stance and plans on Governance, please refer to The 3 Pillars of Polygon Governance post in the community forum. The first pillar can be found here and the second one here. You can follow our updates on Governance here. # aggregator-resistance.md: In the same way that the system cannot reach L2 state finality without an active and well-functioning Sequencer, there can be no finality without an active and well-functioning Aggregator. The absence or failure of the trusted aggregator means that the L2 state transitions are never updated in L1. For this reason, L1 PolygonZkEVM.sol contract has a function named verifyBatches that allows anyone to aggregate sequences of batches. pil function verifyBatches(uint64 pendingStateNum, uint64 initNumBatch, uint64 finalNewBatch, bytes32 newLocalExitRoot, bytes32 newStateRoot, uint256 [2] calldata proofA, uint256 [2] [2] calldata proofB, uint256 [2] calldata proofC,) public ifNotEmergencyState As previously stated, the verifyBatches function accepts the same arguments as the trustedVerifyBatches function. However, the verifyBatches function adds two additional constraints for a sequence to be aggregated, as well as a new L2 pending state. Along with the conditions required in the trustedVerifyBatches function, the following conditions must also be met in verifyBatches: - The contract must not be in an emergency state - A trustedAggregatorTimeout storage variable delay from the timestamp of the last batch in the sequence (when the batch was sequenced) must have been passed. The contract administrator configures the trustedAggregatorTimeout variable. The function verifies the zero-knowledge proof of computational integrity if all conditions are met. However, unlike the trustedVerifyBatches function, if verification is successful, the sequence is not immediately aggregated. Instead, the verified sequence is added to the pendingStateTransitions mapping and is aggregated after a delay specified by the pendingStateTimeout. // pendingStateNumber --> PendingState mapping(uint256 => PendingState) public pendingStateTransitions; The struct used looks like this: struct PendingState { uint64 timestamp; uint64 lastVerifiedBatch; bytes32 exitRoot; bytes32 stateRoot; } Verified batch sequences remain in an intermediate state known as Pending state, where their state transition has not yet been consolidated. While in this state, neither the new L2 state root nor the bridge's new GlobalExitRoot have been added to the batchNumToStateRoot mapping. The lastPendingState storage variable is used to keep track of the number of pending state transitions that need to be consolidated and serves as the mapping's key of entry. The Aggregator receives an aggregation reward once the zero-knowledge proof has been verified. The below figure shows the L2 Stages timeline from a batch perspective, and the actions that triggers its inclusion in the next L2 state stage, when a batch sequence is Aggregated through the verifyBatches function. !L2 State stages timeline with pending state The presence of batch sequences in pending state has no effect on the correct and proper functioning of the protocol. Non-forced batch sequences are verified before pending ones, and not all sequences enter the pending state. The storage variable lastVerifiedBatch keeps track of the index of the most recently verified and aggregated batch. As a result, even if a batch sequence is pretentiously verified, the index of the last verified batch is queried via a function called getLastVerifiedBatch. If there are any pending state transitions, this function returns the index of the last batch in that state; otherwise, it returns the lastVerifiedBatch. function getLastVerifiedBatch() public view returns (uint64) When the sequenceBatches function is called, an attempt is made to consolidate the pending state by calling the internal function tryConsolidatePendingState. If the pendingStateTimeout has elapsed since the pending batches verification, this function consolidates the pending state transitions. There is no need to check the zero-knowledge proof again because it has already been verified. This mechanism is designed to help in the detection of any soundness vulnerabilities that could otherwise be exploited in the zero-knowledge proof verification system, thereby protecting assets from being bridged out to the L2 by malicious actors. # emergency-state.md: The emergency state is a Consensus Contract state (of 'PolygonZkEVM.sol' and 'PolygonZkEVMBridge.sol') that, when activated, terminates batch sequencing and bridge operations. The goal of enabling the emergency state is to allow the Polygon team to solve cases of soundness vulnerabilities or exploitation of any smart contract bugs. It is a security measure used to protect users' assets in zkEVM. The following functions will be disabled while in the emergency state: - sequenceBatches - verifyBatches - forceBatch - sequenceForceBatches - proveNonDeterministicPendingState As a result, while the contract is in the emergency state, the Sequencer cannot sequence batches. Meanwhile, the trusted Aggregator will be able to consolidate additional state transitions or override a pending state transition that can be proven to be non-deterministic. When the same sequence of batches is successfully verified with two different resulting L2 state root values, a non-deterministic state transition occurs. This situation could arise if a soundness vulnerability in the verification of the zero-knowledge proof of computational integrity is exploited. When is the emergency state activated? The emergency state can only be triggered by two contract functions: 1. It can be directly activated by calling the activateEmergencyState function by contract owner. 2. It can also be called by anyone after a HALT AGGREGATION TIMEOUT constant delay (of one week) has passed. The timeout begins when the batch corresponding to the sequencedBatchNum argument has been sequenced but not yet verified. This situation directly implies that no one can aggregate batch sequences. The objective is to temporarily stop the protocol until aggregation activity resumes. function activateEmergencyState(uint64 sequencedBatchNum) external Additionally, anyone can use the proveNonDeterministicPendingState function to trigger the emergency state, but only if they can prove that some pending state is non-deterministic. function proveNonDeterministicPendingState(uint64 initPendingStateNum , uint64 finalPendingStateNum , uint64 initNumBatch , uint64 finalNewBatch , bytes32 newLocalExitRoot , bytes32 newStateRoot , uint256 [2] calldata proofA , uint256 [2][2] calldata proofB , uint256 [2] calldata proofC) public ifNotEmergencyState Overriding a pending state If a soundness vulnerability is exploited, the Trusted Aggregator has the ability to override a non-deterministic pending state. To initiate the override, use the overridePendingState function. Because the Trusted Aggregator is a trusted entity in the system, only the L2 state root provided by the Trusted Aggregator is considered valid for consolidation in the event of a non-deterministic state transition. function overridePendingState(uint64 initPendingStateNum , uint64 finalPendingStateNum , uint64 initNumBatch , uint64 finalNewBatch , bytes32 newLocalExitRoot , bytes32 newStateRoot , uint256 [2] calldata proofA , uint256 [2][2] calldata proofB , uint256 [2] calldata proofC) public onlyTrustedAggregator To successfully override a pending state, the Trusted Aggregator must submit a proof that will be verified in the same way, as in the proveNonDeterministicPendingState function. If the proof is successfully verified, the pending state transition is wiped and a new one is directly consolidated. To summarize, the emergency state can only be activated: - when the contract owner deems it appropriate, or - when aggregation activity is halted due to a HALTAGGREGATIONTIMEOUT, or - when anyone can demonstrate that a pending state is non-deterministic. # sequencer-resistance.md: Users must rely on a trusted sequencer for their transactions to be executed in the L2. However, users can include their transactions in a forced batch if they are unable to execute them through the trusted sequencer. A forced batch is a collection of L2 transactions that users can commit to L1 to publicly declare their intent to execute those transactions. !Forced batch sequencing flow The PolygonZkEVM.sol contract has a forcedBatches mapping, as shown in the above figure, in which users can submit transaction batches to be forced. The forcedBatches mapping serves as an immutable notice board where forced batches are timestamped and published before being included in a sequence. // ForceBatchNum --> hashedForcedBatchData mapping(uint64 => bytes32) public forcedBatches; !!!caution The trusted sequencer include these forced batches in future sequences to maintain its status as a trusted entity. Otherwise, users can demonstrate that they are being censored, and the trusted sequencer's trusted status is revoked. Although the trusted sequencer is incentivized to sequence the forced batches published in the forcedBatches mapping, this does not guarantee finality of the transactions' execution in those batches. In order to ensure finality in the case of trusted sequencer's malfunction, the L1 PolygonZkEVM.sol contract has an alternative batch sequencing function called sequenceForceBatches. This function allows anyone to sequence forced batches that have been published for a time period, specified by the public constant forceBatchTimeout, yet they have not been sequenced. The timeout is set to 5 days. Any user can publish a batch to be forced by directly calling forceBatch function: bash function sequenceForceBatches(ForcedBatchData[] memory batches) public ifNotEmergencyState isForceBatchAllowed in Etrog it is: bash function sequenceForceBatches(BatchData[] calldata batches) external virtual isSenderAllowedToForceBatches In order to successfully publish a forced batch to the forcedBatches mapping, the following conditions must be met, otherwise the transaction reverts; - The contract must not be in emergency state, - The force batches must be allowed, - The maticAmount argument must be higher than the MATIC fee per batch. It is the maximum amount of MATIC tokens the user is willing to pay as a forced batch publication fee. The fee for publishing a forced batch is the same as the fee for sequencing, and it is therefore set in the batchFee storage variable. Since the fee is paid when a forced batch is published, it is not be paid again when the batch is sequenced. - The length of the transactions byte array must be less than the value of MAXTRANSACTIONSBYTELENGTH constant (which is set at 120000). The forced batch is entered in forcedBatches mapping keyed by its force batch index. struct ForcedBatchData { bytes transactions; bytes32 globalExitRoot; uint64 minForcedTimestamp; } The lastForceBatch storage variable, which is incremented for each forced batch published, serves as a forced batch counter and thus provides a specific index number. The value entered is a hash digest of the ABI-encoded packed struct fields from the lastForceBatch. keccak256(abi.encodePacked(keccak256(bytes transactions), bytes32 globalExitRoot, unint64 minTimestamp)); Storage slots (mapping entries) are only used to store a commitment of the forced batch for storage usage optimization reasons. Data availability is ensured because it can be recovered from transaction calldata. The contract sets the minTimestamp to the L1 block timestamp, at which point the forced batch is published. In the extremely unlikely event that the trusted sequencer fails, any user can use the sequenceForceBatches function to sequence forced batches: function sequenceForceBatches(ForcedBatchData[] memory batches) public ifNotEmergencyState isForceBatchAllowed The sequenceForceBatches function is similar to the sequenceBatches function, but it can be called by anyone if batch forcing is enabled. The sequenceForceBatches function determines whether each batch in the submitted sequence has been published to the forcedBatches mapping for a period of time greater than the forceBatchTimeout. Because the MATIC batch fee was paid at the time of publication, it is not required to be paid again. If the sequence of forced batches meets all of the sequence conditions, it is added to the sequencedBatches mapping as a regular one. As a result, a sequenceForceBatches event is generated. event SequenceForceBatches(uint64 indexed numBatch); Note that since the sequences of forced batches, sequenced using the sequenceForceBatches function, never reaches a trusted state, a divergence occurs between the nodeâ€™s local trusted L2 state and the virtual L2 state committed in the L1 PolygonZkEVM.sol contract. This situation has been detected and handled by the node software. It reorganises its local L2 state instance based on the L2 state retrieved from L1. The below diagram depicts the distinction between trusted and virtual L2 states that occur when a forced batch sequence is executed. !Differences between trusted and virtual L2 state # batch-aggregation.md: !!!info This document is a continuation in the series of articles explaining the transaction life cycle inside Polygon zkEVM. The trusted aggregator should aggregate the sequences of batches previously committed by the trusted sequencer in order to achieve the L2 state final stage, which is the consolidated state. Aggregating a sequence means successfully adding the resulting L2 state root to the batchNumToStateRoot mapping in the L1 PolygonZkEVM.sol contract. This is a storage structure that contains all of the consolidated L2 state roots, which are keyed by the last batch index of each aggregated sequence of batches. // BatchNum --> state root mapping (uint64 => bytes32) public batchNumToStateRoot; Furthermore, the aggregationÂ implies the successful verification of the zero-knowledge proof of the computational integrity of the transaction batches execution. A SNARK (succinct non-interactive arguments of knowledge) is the underlying zero-knowledge proof verification schema. One of its key characteristics is the proof's conciseness and speed of verification. As a result, the integrity of an exhaustive computation can be verified using a fraction of the computational resources required by the original computation. As a result, by employing a SNARK schema, we can provide on-chain security to exhaustive off-chain computations in a gas-efficient manner. !Off-chain L2 state transition with on-chain security inheritance As shown in the above diagram, the off-chain execution of the batches supposes an L2 state transition, and consequently, a change to a new L2 state root. A computation integrity (CI) proof of the execution is generated by the aggregator, and its on-chain verification ensures validity of that resulting L2 state root. Aggregating a sequence of batches In order to aggregate a sequence of batches, the trusted aggregator must call the trustedVerifyBatches method: function trustedVerifyBatches (uint64 pendingStateNum, uint64 initNumBatch, uint64 finalNewBatch, bytes32 newLocalExitRoot, bytes32 newStateRoot, uint256 [2] calldata proofA, uint256 [2] [2] calldata proofB, uint256 [2] calldata proofC) public onlyTrustedAggregator â€‹where, - pendingStateNum is the number of state transitions pending to be consolidated, which is set to 0 for as long as the trusted aggregator is in operation. The pendingState functions as a security measure to be used when L2 state is consolidated by an independent aggregator. - initNumBatch is the index of the last batch in the last aggregated sequence. - finalNewBatch is the index of the last batch in the sequence being aggregated. - newLocalExitRoot is the root of the Bridgeâ€™s L2 exit Merkle tree at the end of sequence execution used to compute new global exit root when the sequence is aggregated, and allows bridge claiming transactions to be successfully executed in L1. - newStateRoot is the L2 state root resulting from the execution of the sequence of batches over an older L2 state. - proof(A,B and C) is the zero-knowledge proof. For a sequence of batches to be successfully aggregated, the following conditions must be met: - Aggregation transaction must be sent from the trusted aggregator account. - initNumBatch argument must be the index of an already aggregated batch. That is, it must have an L2 state root in batchNumToStateRoot mapping. - initNumBatch argument must be less or equal to the last aggregated batch index. - The sequence to be aggregated must have at least one batch. - initNumBatch and finalNewBatch arguments have to be sequenced batches, that is, to be present in the sequencedBatches mapping. - Zero-knowledge proof of computational integrity must be successfully verified. The executor and the prover are services of the aggregator node that execute batches and generate zero-knowledge proofs. We herein treat them as Ethereum Virtual Machine black box interpreters that can: - execute a sequence of transaction batches on the current L2 state, - calculate the resulting L2 state root, and - generate a zero-knowledge proof of computational integrity for the execution. The proving/verification system is designed in such a way that successful proof verification equates to cryptographically proving that executing the given sequence of batches over a specific L2 state results in an L2 state represented by the newStateRoot argument. The following code snippet is a part of the PolygonZkEVM.sol contract, which shows the zero-knowledge proof verification: // Get snark bytes bytes memory snarkHashBytes = getInputSnarkBytes (initNumBatch, finalNewBatch, newLocalExitRoot, oldStateRoot, newStateRoot); // Calculate the snark input uint256 inputSnark = uint256(sha256(snarkHashBytes)) % RFIELD; // Verify proof require (rollupVerifier.verifyProof(proofA, proofB, proofC, [inputSnark]), "PolygonZkEVM :: verifyBatches : Invalid proof"); RollupVerifier rollupVerifier is an external contract that has a function verifyProof that takes a proof (proofA, proofB, proofC) and a value inputSnark and returns a boolean value that is true if the proof is valid and false if it isnâ€™t. The successful verification of the proof just confirms the integrity of the computation, but not that the correct inputs were used and that they resulted in the correct output values. Public arguments are used to publicly disclose key points of the computation being proved, in order to prove that it was performed using the correct inputs and reveal the outputs. This way, during the proof verification, the L1 smart contract sets the public arguments to ensure that the state transition being proved corresponds to the execution of the batches committed by the trusted sequencer. inputSnark inputSnark is a 256-bits unique cryptographic representative of a specific L2 state transition, which is used as public argument. It is computed as sha256 mod % RFIELD hash of a bytes string called snarkHashBytes (modulo operator is needed due to math primitives used in SNARKs). snarkHashBytes array is computed by a smart contractâ€™s function called getInputSnarkBytes and it is an ABI-encoded packed string of the following values: - msg.sender: Address of trusted aggregator. - oldStateRoot: L2 state root that represents the L2 state before the state transition that wants to be proven. - oldAccInputHash: Accumulated hash of the last batch aggregated. - initNumBatch: Index of the last batch aggregated. - chainID: Unique chain identifier. - newStateRoot: L2 state root that represents the L2 state after the state transition that is being proved. - newAccInputHash: Accumulated hash of the last batch in the sequence that is being aggregated. - newLocalExitRoot: Root of the bridgeâ€™s L2 exit Merkle tree at the end of sequence execution. - finalNewBatch: Number of the final batch in the execution range. inputSnark represents all the L2 transactions of a specific L2 state transition, executed in a specific order, in a specific L2 (chainID), and proved by a specific trusted aggregator (msg.sender). The trustedVerifyBatches function not only verifies the validity of the zero-knowledge proof, but it also checks that the value of inputSnark corresponds to an L2 state transition that is pending to be aggregated. If the internal call to verifyAndRewardBatches returns true, it means that the sequence of batches is verified successfully, and then the newStateRoot argument is added to the batchNumToStateRoot mapping. The index of the last batch in the sequence is used as the key for the entry. Finally a TrustedVerifyBatches event is emitted. event TrustedVerifyBatches (uint64 indexed numBatch, bytes32 stateRoot, address indexed aggregator); Once the batches have been successfully aggregated in L1, all zkEVM nodes can validate their local L2 state by retrieving and validating consolidated roots directly from the L1 consensus contract (PolygonZkEVM.sol). As a result, the L2 consolidated state has been reached. # batch-sequencing.md: !!!info This document is a continuation in the series of articles explaining the transaction life cycle inside Polygon zkEVM. Batches need to be sequenced and validated before they can become a part of the L2 virtual state. The trusted sequencer successfully adds a batch to a sequence of batches using the L1 PolygonZkEVM.sol contract's sequencedBatches mapping, which is basically a storage structure that holds the queue of sequences defining the virtual state. // SequenceBatchNum --> SequencedBatchData mapping(uint64 => SequencedBatchData) public sequencedBatches; Batches must be a part of the array of batches to be sequenced in order to be sequenced. The trusted sequencer invokes the PolygonZkEVM.sol contract, which employs its sequenceBatches mapping, which accepts an array of batches to be sequenced as an argument. Please see the code snippet provided below. function sequenceBatches (BatchData[] memory batches) public ifNotEmergencyState onlyTrustedSequencer The below figure shows the logic structure of a sequence of batches. !An outline of sequenced batches Max & min batch size bounds The contract's public constant, MAXTRANSACTIONSBYTELENGTH, determines the maximum number of transactions that can be included in a batch (120000). Similarly, the number of batches in a sequence is limited by the contract's public constant MAXVERIFYBATCHES (1000). The batches array must contain at least one batch and no more than the value of the constant MAXVERIFYBATCHES. Only the trusted sequencer's Ethereum account can access the sequencedBatches mapping. The contract must not be in an emergency state. The function call reverts if the above conditions are not met. Batch validity & L2 state integrity The sequencedBatches function iterates over every batch of the sequence, checking its validity. A valid batch must meet the following criteria: - It must include a globalExitRoot value that is present in the GlobalExitRootMap of the bridgeâ€™s L1 PolygonZkEVMGlobalExitRoot.sol contract. A batch is valid only if it includes a valid globalExitRoot. - The length of the transactions byte array must be less than the value of MAXTRANSACTIONSBYTELENGTH constant. - The timestamp of the block must be greater or equal to that of the last block (of a sequenced batch), but less than or equal to the timestamp of the block where the sequencing L1 transaction is executed. All blocks must be ordered by time. If one block is not valid, the transaction reverts, discarding the entire sequence. Otherwise, if all blocks in the batches that are to be sequenced are valid, the sequencing process continues as normal. A storage variable called lastBatchSequenced is used as a batch counter, and it is thus incremented each time a batch is sequenced. It gives a specific index number to each batch that is used as a position value in the batch chain. The same hashing mechanism used in blockchains to link one block to the next is used in batches to ensure the cryptographic integrity of the batch chain. That is, including the previous batch's digest among the data used to compute the next batch's digest. As a result, the digest of a given batch is an accumulated hash of all previously sequenced batches, hence the name accumulated hash of a batch, denoted by oldAccInputHash for the old and newAccInputHash for the new. An accumulated hash of a specific batch has the following structure: c++ keccak256 (abi.encodePacked (bytes32 oldAccInputHash, keccak256(bytes transactions), bytes32 globalExitRoot , uint64 timestamp , address seqAddress)) â€‹where, - oldAccInputHash is the accumulated hash of the previous sequenced batch. - keccack256(transactions) is the Keccak digest of the transactions byte array. - globalExitRoot is the root of the bridgeâ€™s global exit Merkle tree. - timestamp is the batch timestamp. - seqAddress is the address of the batch sequencer. !Batch chain structure As shown in the diagram above, each accumulated input hash ensures the integrity of the current batch's data (i.e., transactions, timestamp, and globalExitRoot, as well as the order in which they were sequenced. It is important to note that any change in the batch chain causes all future accumulated input hashes to be incorrect, demonstrating a lack of integrity in the resulting L2 state. The batch sequence is added to the sequencedBatches mapping using the following SequencedBatchData struct only after the validity of all batches in a sequence has been verified and the accumulated hash of each batch has been computed. struct SequencedBatchData { bytes32 accInputHash; uint64 sequencedTimestamp; uint64 previousLastBatchSequenced; } â€‹where, - accInputHash is the batchâ€™s a unique cryptographic finger-print of the last batch in the sequence. - sequencedTimestamp is the timestamp of the block where the sequencing L1 transaction is executed. - previousLastBatchSequenced is the index of the last sequenced batch before the first batch of the current sequence (i.e., the last batch of the previous sequence). The index number of the last batch in the sequence is the key, and the SequencedBatchData struct is the value, in the sequencedBatches mapping. Batch data minimal storage Since storage operations on L1 are very costly in terms of gas consumption, it is essential to use them as sparingly as possible. To accomplish this, storage slots (or mapping entries) are used solely to store a sequence commitment. Each mapping entry commits two batch indices, - Last batch of the previous sequence as value of SequencedBatchData struct, - Last batch of the current sequence as mapping key, along with the accumulated hash of the last batch in the current sequence and a timestamp. It is important to note that only the accumulated hash of the last batch in the sequence is saved; all others are computed on the fly in order to obtain the last one. As previously stated, the hash digest becomes a commitment of the entire batch chain. Batch indices also commit useful information like the number of batches in the sequence and their position in the batch chain. The timestamp anchors the sequence to a specific point in time. The data availability of the L2 transactions is guaranteed because the data of each batch can be recovered from the calldata of the sequencing transaction, which is not part of the contract storage but is part of the L1 state. Finally a SequenceBatches event emits. solidity event SequenceBatches (uint64 indexed numBatch) Once the batches are successfully sequenced in L1, all zkEVM nodes can sync their local L2 state by fetching the data directly from the L1 PolygonZkEVM.sol contract, without having to rely on the trusted sequencer alone. This is how the L2 virtual state is reached. # index.md: !!! info Users need funds on L2 to be able to send transactions to the L2 network. To do so, users need to deposit Ether to L2 through the Polygon Portal. Bridging: - Deposit ETH. - Wait until globalExitRoot is posted on L2. - Perform claim on L2 and receive the funds. Transaction process on L2 and the three (3) states: - User initiates a transaction using their wallet (e.g. MetaMask) and sends it to a trusted sequencer. - The transaction gets finalized on L2 once the trusted sequencer commits to adding the transaction to a batch. This is known as the trusted state. - Sequencer sends the batch data to L1 smart contract, enabling any L2 node to synchronize from L1 in a trustless way. This is also known as the virtual state. - Aggregator takes the pending transactions, and builds a proof. - Once the proof is verified, the transactions attain L1 finality (important for withdrawal of funds from L2). This is called the consolidated state. The above process is a summarized version of how transactions are processed in Polygon zkEVM. Take a look at the complete description in the transaction life cycle # submit-transaction.md: !!!info This series of documents describes in detail the various stages L2 transactions go through, from the time they are created in users' wallets to the time they are finally verified with indisputable evidence on L1. Submit transaction Transactions in the Polygon zkEVM network are created in users' wallets and signed with their private keys. Once generated and signed, the transactions are sent to the trusted sequencer's node via their JSON-RPC interface. The transactions are then stored in the pending transactions pool, where they await the sequencer's selection for execution or discard. Users and the zkEVM communicate using JSON-RPC, which is fully compatible with Ethereum RPC. This approach allows any EVM-compatible application, such as wallet software, to function and feel like actual Ethereum network users. Transactions and blocks on zkEVM In the current design, a single transaction is equivalent to one block. This design strategy not only improves RPC and P2P communication between nodes, but also enhances compatibility with existing tooling and facilitates fast finality in L2. It also simplifies the process of locating user transactions. # transaction-batching.md: !!!info This document is a continuation in the series of articles explaining the transaction life cycle inside Polygon zkEVM. The trusted sequencer must batch the transactions using the following BatchData struct specified in the PolygonZkEVM.sol contract: struct BatchData { bytes transactions; bytes32 globalExitRoot; uint64 timestamp; uint64 minForcedTimestamp; } Transactions â€‹These are byte arrays containing the concatenated batch transactions. â€‹Each transaction is encoded according to the Ethereum pre-EIP-155 or EIP-155 formats using RLP (recursive-length prefix) standard, together with the signature values, v, r and s, concatenated as shown below; 1. EIP-155: $\mathtt{\ rlp(nonce, gasprice, gasLimit, to, value, data, chainid, 0, 0,) \#v\#r\#s\#effectivePercentage}$ 2. pre-EIP-155: $\mathtt{\ rlp(nonce, gasprice, gasLimit, to, value, data) \#v\#r\#s\#effectivePercentage}$ GlobalExitRoot The root of the bridge's global exit Merkle tree, called GlobalExitRoot, is synchronized in the L2 state at the start of batch execution. The bridge transports assets between L1 and L2, and a claiming transaction unlocks the asset in the destination network. Timestamp â€‹In as much as Ethereum blocks have timestamps, and since the Etrog upgrade, each block has its own timestamp. â€‹There are two constraints each timestamp must satisfy in order to ensure that blocks are ordered in time and synchronized with L1 blocks: 1. The timestamp of a given block must be greater or equal to that of the last block (in a sequenced batch). 2. The maximum block timestamp a trusted sequencer can set to a block is the timestamp of the block where the sequencing L1 transaction is executed. MinForcedTimestamp If a batch is a so-called forced batch, the MinForcedTimestamp parameter must be greater than zero. Censorship is mitigated by utilizing forced batches. Further details on this is provided in the following sections. # transaction-execution.md: !!!info This document is a continuation in the series of articles explaining the transaction life cycle inside Polygon zkEVM. The trusted sequencer reads transactions from the pool and decides which transactions to order and execute. Once executed, transactions are added to blocks, then the blocks fill batches, and the sequencer's local L2 state is updated. Once a transaction is added to the L2 state, it is broadcast to all other zkEVM nodes via a broadcast service. It is worth noting that by relying on the trusted sequencer, we can achieve fast transaction finality (faster than in L1). However, the resulting L2 state remains a trusted state until the batch is committed in the consensus contract. Verification on layer 1 Users typically interact with trusted L2 state. However, because of specific protocol characteristics, the verification process for L2 transactions on L1 (to enable withdrawals) can be lengthy, usually taking about 30 minutes, and in rare instances, up to a week. !!!note - What is the rare case scenario? Verification of transactions on L1 can take 1 week in the case when an emergency state is activated or the aggregator does not batch any proofs at all. - Additionally, the emergency mode is activated if a sequenced batch is not aggregated in 7 days. Please refer to this guide to understand more about the emergency state. As a result, users should be mindful of the potential risks associated with high-value transactions, particularly those that cannot be reversed, such as off-ramps, over-the-counter transactions, and alternative bridges. # aggregator-proofs.md: In this document, we delve into a crucial component of the zkEVM called the aggregator. The aggregator takes several proofs, each attesting to correct execution of some batch, and combines them into a single proof. Verifying the combined proof is equivalent to confirming the accuracy of all individual proofs for each batch. The mechanisms used by the aggregator to generate such an aggregated proof are known as proof recursion and proof aggregation. Both mechanisms aim at increasing the system throughput. This document: - Provides some insight on how both the proof recursion and proof aggregation mechanisms work. - Discusses topics such as the "prove anything" paradigm, which is a mechanism that allows proofs of any input to be generated regardless of whether it is erroneous or not. And, of course, since an invalid input should not cause a state change, the corresponding proof attests to the absence of a state change rather than its presence. - Explores the zkCounters concept, which refers to a mechanism used in the zkEVM to measure batch sizes and ensures that only batches that fit into execution traces are allowed. In the event of a batch size exceeding set limits, the system should throw an out of counters (OOC) error. - Concludes with a discussion on how to eliminate zkCounters in the future, achievable by implementing a method known as Variable Degree Composite Proofs (VADCOPs). Prove anything paradigm To address the potential threat of malicious sequencers, we adopt the â€œprove anything" paradigm. With this approach, the prover possesses the capability to generate a proof of execution for any input data. But this is on the condition that each batch must maintain a bounded amount of data. The smart contract ensures compliance with this requirement throughout the sequencing process. Batch execution is carried out for any input data, resulting in the generation of a proof that confirms a state change for valid input data, or no state change for invalid input data, as depicted in the figure below. This strategy guarantees robust validation of execution outcomes, and provides a reliable mechanism to handle potential, malicious behavior. !Figure: Correct vs. wrong state change Invalid transactions Let us describe some errors in transactions that cause the state to remain unchanged, as shown in the figure below. Reverted transaction A transaction may revert during execution due to many reasons, such as: - Running out of gas. - Having a stack that is too large. - Encountering a revert call in the code. This is a common scenario in EVM processing. Invalid intrinsic transaction An invalid intrinsic transaction is a transaction that cannot be processed, and thus has no impact on the current state. Keep in mind that this transaction could be part of a virtual batch. Examples of errors in this scenario are: incorrect nonce, insufficient balance, etc. The zkEVM's trusted sequencer is unlikely to input an incorrect nonce. However, any member of the community can submit a batch, which may result in an error. !Figure: Invalid or reverted txs Invalid batches Some of the errors that occur at the batch level can cause the state to remain unchanged. As depicted in the figure below, such errors might occur due to invalid data, or exhaustion of prover resources. - Invalid Data: Failure to decode RLP-encoded transactions, that is having garbage input. - Prover resource exhaustion: The zkEVM prover manages resources in terms of row counters in the Stack, also known as zkCounters. Processing a batch may fail due running out of zkCounters (OOC). !Figure: State not updaated The above figure shows that when batch processing fails, the state remains unchanged, $S^{L2x}_{i+1} = S^{L2x}_{i}$, and a proof of no state change is produced. Although this occurrence is infrequent, it is possible. The prove anything approach allows the system to implement an anti-censorship measure called forced batches. That is, in the case where the trusted sequencer does not process a user's transactions, the user can take the role of a sequencer by taking their L2 transactions into the virtual state. The main use case is to allow a user to send bridge transactions to withdraw assets from L2 without the risk of censorship, which would otherwise make fund withdrawals impossible. Since every user who sends L2 batch data is, by default, 'untrusted,' the system must ensure that anything sent by any user can be verified. The forced batches mechanism is elaborated on in the Malfunction resistance subsection. zkCounters Counters are used during the execution of a batch to monitor the row count in each state machine, including the main state machine. The management of these counters is incorporated within the computation process. This way, if a computation exhausts its assigned resources while generating the proof, the system can verify and prove that the batch does not cause a state change. As mentioned earlier, the issue of running out of assigned resources is referred to as an out of counters (OOC) error. This error is specific to the Polygon zkEVM state machine design, where the execution trace has a fixed number of rows. The figure below depicts a simplified scenario of the OOC error for a Keccak state machine, where its arithmetization permits only up to four Keccak operations before exhausting the available rows. So, an OOC error occurs if a transaction invokes five or more Keccak operations. !Figure: Executor ... possible keccaks VADCOPs Although the current version of the proving system has this limitation in the backend (forcing execution traces of all state machines to have the same amount of rows), it is expected to be resolved with the implementation of a proving technique called variable degree composite proofs (VADCOPs). VADCOPs are designed to partition large execution traces with many rows into smaller execution traces with fewer rows. The main advantage with VADCOPs is the elimination of zkCounters. See the figure below for a simplified illustration of the basic idea of what VADCOPs can achieve. In this scenario, it becomes feasible to execute five Keccak operations despite the limit of executing only four operations at a time. At a high level, the solution involves splitting the proof into two parts, each containing fewer rows, and then aggregating them to demonstrate the execution of five Keccak operations. !Figure: VADCOPs with Keccak The ongoing development of VADCOPs includes rewriting of both the cryptographic backend and the constraint language called Polynomial Identities Language version 2 (PIL2). Discussing the proof The preceding subsection has briefly outlined what VADCOPs can do. The concept of VADCOPs suggests that we can somehow aggregate proofs together. Even though aggregating proofs requires more effort from the prover, it allows many proofs to be verified simultaneously. This scenario is of interest to us because it creates a single, unified proof that allows various batches to be verified simultaneously, thereby increasing system throughput. Two techniques of interest in this area are proof recursion and proof aggregation. Proof recursion Recursion, which is also called compression, in proof systems means the prover deals with smaller, quicker-to-verify proofs instead of large, time-consuming proofs. Essentially, the prover generates a proof at the current stage that attests to the fact that the verification of the previous stage was correctly performed. At the end of the proof recursion, a SNARK proof is generated to attest to the correct verification of each final proof. Consequently, since the succinctness property of SNARKs dictates that verification time should be relatively smaller than the proving time, the final SNARK proofs are much smaller and faster to verify than the proofs in the initial stages. The figure below illustrates the core principle of proof recursion. !Figure: Inner and Outer proving system The "Outer Prover", seen in the figure above, is responsible for proving the correct verification of the proof π{big} and it outputs the proof π{small} which is sent to the "Outer Verifier". The proof π{small} being a SNARK proof is therefore much smaller compared to π{big} due to the succinctness property of SNARKs. Notice that the set of public values vary from one recursion stage to the next. Proof aggregation In the zkEVM context, aggregation is a technique that allows the prover to generate a single proof that covers multiple L2 batches. This reduces the number of proofs to be verified, and thus increases the system throughput. Proof aggregation means sending a single L1 transaction that aggregates multiple batches, and this improves the batch consolidation rate. Note that the proving system limits aggregation to consecutive batches only. Also, we can aggregate single-batch proofs with multiple-batch proofs, as shown in the figure below. This is achievable because of a technique, used in the cryptographic backend, called normalization. Finally, we remark that the smart contract also sets limits to the number of batches that can be aggregated. !Figure: zkEVM recursion and aggregation We now provide concrete blocks and steps used to prove correct execution of several batches by the Polygon zkEVM, using recursion and aggregation. An overview of the overall process is depicted in the figure below. !Figure: Specific recursion design The first proving system generates such a big proof since it has a lot of high degree polynomials. Thereafter, the Compression Stage is invoked and applied on in each batch proof, aiming to reduce the number of polynomials used, and hence allowing reduction in the proof size. Next, the Normalization Stage is invoked, allowing each aggregator verifier and the normalization verifier to be exactly the same, permitting successful aggregation via a recursion. Once the normalization step has been finished, the proofs go through aggregation. The procedure is to construct a binary tree of proofs by aggregating every pair of proofs into one. We call this step, the Aggregation Stage. In this step, two batch proofs are put together into one, and this repeated for as long as there's more than one proof to aggregate. Observe that the Aggregation Stage is designed to accept: - Two compressed proofs. - Two aggregated proofs. - One compressed proof and an aggregated proof. The Final Stage is the last STARK step of the recursion process, which is in charge of verifying a proof over a finite field associated with the BN128 elliptic curve. This field is completely different from the Goldilocks field $\mathbb{F}p$, where $p = 2^{64} - 2^{32} + 1$, which is used in all earlier stages of the proof recursion and aggregation process. The SNARK Stage converts the STARK proof, which is the output of the Final Stage, to an FFLONK proof. The final output of the proof recursion and aggregation process is an FFLONK proof, which is a SNARK, and it gets verified via the ec-pairing precompiled contract. Introducing the aggregator In the Polygon zkEVM architecture, the aggregator is the component responsible for performing proof aggregation. Its role is to aggregate several proofs into one, and send the aggregated proof to the L1 Smart Contract through the $\texttt{verifyBatches()}$ function. The aggregator invokes the $\texttt{verifyBatches()}$ function on the smart contract, passing the following parameters: - The initial batch number, $\texttt{initNumBatch}$. - The final batch number, $\texttt{finalNewBatch}$. - The root, $\texttt{newStateRoot}$. - The aggregated proof, $\mathbf{\pi{a,b,c,...}}$. See the figure below, depicting a single, aggregated proof, $\pi{a,b,c,...}$, as an input to the verifier smart contract. The previous root is stored in the smart contract, eliminating the need to transmit it. Recall that the smart contract contains a summary of the batch information in the accumulated input hash. !Figure: Aggregation and L1 smart contract As seen in the figure above, provers link up with the aggregator to send their proofs. The aggregator operates as a network server, establishing connections with provers that function as network clients. Thus, it functions as a server, responsible for determining how to horizontally scale provers to achieve an optimal batch consolidation rate. Scaling is essential to prevent a buildup of additional batches awaiting consolidation. The aggregator also keeps a record of authorized provers. Both the aggregator and the provers operate in a cloud-based environment, with the provers configured as high-resource instances. See the figure below, for such a cloud-based environment. The configuration allows for effective and scalable control over evidence processing, ensuring the system can handle various workloads while maintaining an efficient batch consolidation rate. !Figure: Cloud-based aggregator and L1 SC Proof inputs and outputs The proof generation process requires several inputs to ensure its soundness: - The aggregator address, serves as a safeguard against malleability in the $\texttt{verifyBatches()}$ function, ensuring that no one can use another aggregatorâ€™s proof. - The previous state root (oldStateRoot), which is already included in the smart contract and does not require explicit sending. - The previous accumulated input hash ($\texttt{oldAccInputHash}$). - The initial batch number ($\texttt{initNumBatch}$). - The chainID and the forkID ensure that the proof is valid only within the intended chain and version of the zkEVM. The figure below, depicts the aggregator inputs and outputs. !Figure: Public inputs to Aggregator The aggregator generates the following outputs: - The updated state root, $\texttt{newStateRoot}$. - The new accumulated input hash, $\texttt{newAccInputHash}$. - The aggregated proof, π. - The batch number of the last batch included in the aggregated proof once it has been successfully generated ($\texttt{finalNewBatch}$). Currently, all the inputs and outputs are public. And this presents an efficiency problem, which we discuss below. The smart contract structure shown in the figure below, enables upgradeability or alteration of the verifier, as necessary. Currently we only provide a verifier called FflonkVerifier.sol. !Figure: zkEVM.sol and FFLONK verifier In this approach, the verifier must utilize a list of publics. But, a verifier with a single input is more cost-effective. In Groth16, for instance, there is one scalar multiplication operation per public input, which costs approximately $10,000$ gas units per public. The key strategy is therefore to: - Create a single public input, which is the hash of all the previous public keys. - Transform all other public inputs into private inputs. The figure below depicts this configuration, where public inputs are in green while private inputs are in red. !Figure: inputSNARK to Aggregator Hence, for the outer proof, there is a single public input in the aggregator called $\texttt{inputSnark}$, which is a Keccak hash of all the previous public inputs and outputs. Therefore, when ZkEVM.sol contract interacts with the VerifierRollups.sol contract, only a single public parameter is passed, minimizing data transmission costs. !Figure: Verifier contract redesign This optimized approach requires two verifications: - Checking that the hash of all private inputs, arranged in the correct order, corresponds to the given $\texttt{inputSnark}$. - Verifying that the total input hash, calculated for all processed batches by the aggregator, matches the specified $\texttt{newAccInputHash}$. # exec-selector-columns.md: In the above discussions, we have generally signalled the operation applied to a specific row by writing the operation alongside that row, but outside the matrix. There is a need to pull in this information into the execution matrix. For this purpose extra columns, called selector columns, are appended to the execution matrix. A selector column will mostly consist of zeros, "0"s, and ones, "1"s, where a 1 appears in the row to which the operation is applied. Example: (Selector columns) Consider the two operations defined in the previous example: $$ \begin{aligned} &\texttt{Op1}:a' = a + b + c \\ &\texttt{Op2}:c' = a + b + c + a' + b' \end{aligned} $$ Their corresponding selector columns are given the same name, and are incorporated in the execution matrix as shown below. $$ \begin{aligned} \begin{array}{|c|c|c|c|c|}\hline {\texttt{ A }} & {\texttt{ B }} & {\texttt{ C }} & {\texttt{OP1}} & {\texttt{OP2}} \\ \hline a0 & b0 & c0 & 1 & 0 \\ \hline a1 & b1 & c1 & 0 & 1 \\ \hline a2 & b2 & c2 & - & - \\ \hline \end{array} \end{aligned} $$ This way, the appearance of 1 in each of the columns $\texttt{Op1}$ and $\texttt{Op2}$ flags when the respective operation is executed. Designing the execution traces in this way is fully aligned with how interpolation is applied to the execution traces, that is to say column-wise. Selector columns are used to control whether the constraints of an operation apply or not, meaning whether $\texttt{Opx}$ or $\texttt{Opy}$ is applied to a particular row. Selector columns and constraints Next we explain how to test the correctness of the execution trace in each row with just one equation. Firstly, note that: $$ \begin{aligned} &\texttt{Op1}:\ a' = a + b + c \quad\quad \Rightarrow \quad a + b + c - a' = 0 \\ &\texttt{Op2}:\ c' = a + b + c + a' + b' \quad \Rightarrow \quad a + b + c + a' + b' - c' = 0 \end{aligned} $$ Secondly, correct execution of each operation is tested with a zero-check. That is, checking each of the second equations: $$ \begin{aligned} &\texttt{Op1}:\ a + b + c - a' = 0 \\ &\texttt{Op2}:\ a + b + c + a' + b' - c' = 0 \end{aligned} $$ Thirdly, each operation is only checked if it was applied to the particular row. That is, only if a value $\texttt{Opx} = 1$ appears in the corresponding row. That is, with respect to each operation $\texttt{Opx}$, the following factor is tested: $$ \texttt{Op1} \cdot \big(1 - \texttt{Op2} \big) = 1\quad \text{only if}\quad \texttt{Op1} = 1 $$ $$ \texttt{Op2} \cdot \big(1 - \texttt{Op1} \big) = 1\quad \text{only if}\quad \texttt{Op2} = 1 $$ Note that, $\texttt{Op1} = 0$ if $\texttt{Op2} = 1$, and conversely, $\texttt{Op2} = 0$ if $\texttt{Op1} = 1$. Putting all these together, checking correctness of execution culminates in testing the following constraint: $$ \texttt{Op1}Â·(1â^' \texttt{Op2})Â·(a+b+câ^'a')+ \texttt{Op2}Â·(1â^' \texttt{Op1})Â·(a+b+c+a'+b'â^'c')=0 $$ Example: (Checking execution correctness) Consider the above execution trace of the operations $\texttt{Op1}$ and $\texttt{Op2}$. In the first row, $\texttt{Op1} = 1 and $\texttt{Op2}$ = 0, so the right-hand side of the constraint reduces to: $$ 1Â·(1â^'0)Â·(a + b + c â^' a') + 0Â·(1 â^' 1)Â·(a + b + c + a' + b' â^' c')= a+b+câ^'a' $$ which equals 0 only if $a' = a + b + c$. And thus proving that $\texttt{Op1}$ was correctly executed. Similarly, in the second row, $\texttt{Op2} = 1 and $\texttt{Op1} = 0, and the right-hand side of the constraint yields: $$ 0Â·(1â^'1)Â·(a + b + c â^' a') + 1Â·(1 â^' 0)Â·(a + b + c + a' + b' â^' c')= a + b + c + a' + b' â^' c' $$ which is 0 only if $c' = a + b + c + a' + b'$. And this proves that $\texttt{Op2}$ was correctly executed. The values in the selector columns are independent of the input values $x = (x0, x1, x2, ... , xl)$, but are rather determined by the computation itself. That is, the input program: $\big(\texttt{Op1}, \texttt{Op2} \big)$, as seen in the above example. Note that the input program could well be $\big(\texttt{Op1}, \texttt{Op1}, \texttt{Op2}, \texttt{Op1}, \texttt{Op2}, \texttt{Op1} \big)$, resulting in the execution trace with the following selector columns: $$ \begin{aligned} \begin{array}{|c|c|c|c|c|}\hline {\texttt{ A }} & {\texttt{ B }} & {\texttt{ C }} & {\texttt{OP1}} & {\texttt{OP2}} \\ \hline a0 & b0 & c0 & 1 & 0 \\ \hline a1 & b1 & c1 & 1 & 0 \\ \hline a2 & b2 & c2 & 0 & 1 \\ \hline a3 & b3 & c3 & - & - \\ \hline a4 & b4 & c4 & 1 & 0 \\ \hline a5 & b5 & c5 & 0 & 1 \\ \hline a6 & b6 & c6 & - & - \\ \hline a7 & b7 & c7 & 1 & 0 \\ \hline \end{array} \end{aligned} $$ Therefore, for each computation, selector columns can be preprocessed. # exec-trace-and-zkevm.md: In our current cryptographic backend, the dimensions of the execution trace are predetermined. However, we don't know beforehand the exact EVM opcodes that will be executed. Consequently, zkEVM operations to be executed are not known. This is so because these opcodes depend on the particular transactions included in L2 batches. Every transaction consists of multiple EVM opcodes, and each EVM opcode corresponds to multiple zkEVM opcodes (i.e., The instructions that are in charge of what values are used to populate the execution trace). This dependence is depicted in the figure below. !Figure: So the predetermined dimensions of execution traces do not help predict the opcodes, but only imply the amount of computation that can be done. In the zkEVM's case, that's the amount and type of L2 transactions for which we can generate a proof. It is therefore hard, in general, to optimize the exact shape or format of a single execution trace matrix. 1. Narrow matrices may easily hit the maximum row limit: - Increasing the number of rows is delicate because the number of rows is a power of 2. - We can only double the size of the matrix, not add extra rows one by one. - So addition should be done with caution, making sure that (as far as possible) not to double the size of the matrix unnecessarily. 2. Wide matrices might be inefficient: - Wide matrices may have too many unused cells and, furthermore be inefficient for mixing many different instructions. - Adding more columns also increases proving time. # execution-trace-design.md: In this section we discuss some aspects pertaining to the shapes or dimensions of the execution traces. The intention is to ensure that each row of an execution trace contains all data required to validate a single or part of a zkASM operation. Notation Since execution traces are created in the context of state machines, columns of an execution trace are also referred to as registries. A registry $\texttt{A}$ is denoted by $\texttt{A} = (a0, a1, ... , a{2^{k}-1})$, where each $a{i+1}$ is the value in $\texttt{A}$ subsequent to ai. Denote this next value in $\texttt{A}$ by a'. That is, $a' = a{i+1}$. Each a' is typically the output of some operation $\texttt{Op}$, applied on some entries $(a{i}, b{i}, c{i})$ from columns $\texttt{A}$, $\texttt{B}$ and $\texttt{C}$. That is: $$ a' = \texttt{Op}\big(a{i}, b{i}, c{i} \big) $$ Example: (Fitting all variables in few columns) Consider the three operations, $\texttt{Op1}$, $\texttt{Op2}$ and $\texttt{Op3}$, as instructions to the executor, written in zkASM. These operations, $\texttt{Op1}$, $\texttt{Op2}$ and $\texttt{Op3}$, are defined to change the next entry of the registry $\texttt{A} = (a0, a1, ... , a{2^{k}-1})$ as follows: $$ \begin{aligned} &\texttt{Op1}:a' =a+b+c\\ &\texttt{Op2}:a' =a+b+c+d+e \\ &\texttt{Op3}:a' =a+b+c+d+e+f+g+h \end{aligned} $$ These constraints define how each of the operations works in the execution trace. A naÃ¯ve construction of the execution trace would yield the following matrix of 8 columns, corresponding to the variables a, b, c, d, e, f, g, h. For illustration purposes, we set each variable to 1. $$ \begin{aligned} \begin{array}{|c|c|c|c|c|c|c|c|}\hline {\texttt{ A }} & {\texttt{ B }} & {\texttt{ C }} & {\texttt{ D }} & {\texttt{ E }} & {\texttt{ F }} & {\texttt{ G }} & {\texttt{ H }}\\ \hline \text{1} & \text{1} & \text{1} & \text{} & \text{} & \text{} & \text{} & \text{} \\ \hline \text{3} & \text{1} & \text{1} & \text{1} & \text{1} & \text{} & \text{} & \text{}\\ \hline \text{7} & \text{1} & \text{1} & \text{1} & \text{1} & \text{1} & \text{1} & \text{1} \\ \hline \text{14} & \text{} & \text{} & \text{} & \text{} & \text{} & \text{} & \text{} \\ \hline \end{array} \end{aligned} $$ Observe that the above execution trace has too many unused cells (15 unused cells in this case). Suppose known research on optimal matrix sizes recommends execution traces with only 6 columns. And we therefore want to create execution traces with only 6 columns. We will later on introduce a third approach involving look-ups. Question: Can all computations involving operations $\texttt{Op1}$, $\texttt{Op2}$ and $\texttt{Op3}$ fit in a 6-column matrix? If so, how can this be done? Caveat: The number of rows must always equal a power of 2. Optimization strategy One possible strategy is to keep the number of rows at 4 , and find appropriate cells in which to store the values of the variables g and h. The simplest solution is to store the values of the variables g and h in the fourth row and columns $\texttt{B}$ and $\texttt{C}$, and hence redefine $\texttt{Op3}$ as: $$ \texttt{Op3}:\ a' =a+b+c+d+e+f+b'+c' $$ See how the above alterations in $\texttt{Op3}$ affects the execution trace. !Figure: In the above table, - The yellow cells represent the operands of $\texttt{Op}$â€‹3. - The green cell represents the output of $\texttt{Op}$3. i.e., The sum of all the values in the yellow cells. This time the execution trace has 7 unused cells, which is a big improment compared to the previous 8-column matrix. Although the columns have reduced, the computation remained the same, and hence the final outcome is also the same. The input program is still $\big(\texttt{Op1}, \texttt{Op2}, \texttt{Op3} \big)$. So setting each variable to 1 yields the matrix below. $$ \begin{aligned} \begin{array}{|c|c|c|c|c|c|}\hline {\texttt{ A }} & {\texttt{ B }} & {\texttt{ C }} & {\texttt{ D }} & {\texttt{ E }} & {\texttt{ F }} \\ \hline \text{1} & \text{1} & \text{1} & \text{} & \text{} & \text{} \\ \hline \text{3} & \text{1} & \text{1} & \text{1} & \text{1} & \text{} \\ \hline \text{7} & \text{1} & \text{1} & \text{1} & \text{1} & \text{1} \\ \hline \text{14} & \text{1} & \text{1} & \text{} & \text{} & \text{} \\ \hline \end{array} \end{aligned} $$ Example: (Compact execution trace for 2 operations) Suppose now that the computation involves only two operations; $\texttt{Op1}$ and $\texttt{Op2}$. Let's examine how to optimize the proving system when the executor's input program is: $$ \big(\texttt{Op1}, \texttt{Op}2 \big) $$ Using a 6-column matrix yields the following execution matrix: !Figure: The blue cells, in the above table, represent the unused cells. These are 9 in number, constituting exactly 50% of all the cells of the matrix. Such an under utilization of the matrix space is dissatisfactory. Optimization strategy An optimal usage of the matrix space can be attained via a strategy similar to the one used in the previous example. That is, placing the last two operands of $\texttt{Op}2$ in the last row of the execution matrix. And thus reduce the number of columns from 6 to 3. $\texttt{Op}2$ is therefore redefined as follows: $$ \texttt{OP2}:\ c' = a + b + c + a' + b' $$ The computation's final outcome is in this case placed in column $\texttt{C}$, as shown in the table below. !Figure: This revised layout ensures that no cell remains unused, reaching the maximum utilization of the matrix space. In conclusion, we note that the number of unused cells strongly depends on the executed instructions and the number of columns of the execution matrix. # intro-proving-sys.md: The zkEVM prover is a software component with which the zkEVM generates proofs that attest to correct execution of programs given a specific set of inputs. While the process of generating a proof is resource-intensive, the time required to verify the proof is significantly shorter, enabling verification to be carried out by a smart contract. On a high level, the prover takes the last batch and the current state as inputs in order to calculate the new state, together with a proof that attests to the computational integrity of the state transition. The figure below depicts a typical state transition $\big(S{i}^{L2x} \to S{i+1}^{L2x}\big)$, - Taking as inputs, a batch $\mathtt{Batch}i$ and the current L2 state $S{i}^{L2x}$. - And the output as a new L2 state $S{i+1}^{L2x}$, together with a proof $\pi{i+1}$. !Figure: Input state and batch, output new state and proof The initial step to produce a proof involves creation of an execution matrix, which is a matrix that records all intermediate computations constituting a larger computation. Such a matrix is also called an execution trace. This larger computation can be thought of as the state transition function, while the smaller intermediate computations are like the zkEVM instructions or opcodes. As we will later on see, it may take one or more zkEVM opcodes to implement a single EVM opcode. This indicates the lack of a strict one-to-one correspondence between EVM opcodes and the Polygon zkEVM's. Typical execution matrix Unless otherwise stated, we use a matrix with three columns; $\texttt{A}$, $\texttt{B}$ and $\texttt{C}$. Furthermore, suppose the number of rows is bounded by some constant N, which we will call length of the execution trace. The columns of an execution trace are often called registers. So, the terms "column" and "register" are used interchangeably in this document. We depict an execution trace of length N, having three registers $\texttt{A}$, $\texttt{B}$ and $\texttt{C}$, in the figure below. $$ \begin{aligned} \begin{array}{|c|c|c|}\hline \bf{\texttt{ A }} & \bf{\texttt{ B }} & \bf{\texttt{ C }}\\ \hline a0 & b0 & c0 \\ \hline a1 & b1 & c1 \\ \hline \dots & \dots & \dots \\ \hline aN & bN & cN \\ \hline \end{array} \end{aligned} $$ A toy example (Execution matrix) Suppose we are given a set of three input values, $x := (x0, x1, x2)$, and we want to use an execution trace to model the following computation: $$ [(x0 +x1)Â·4]Â·x2 $$ Let us also suppose that the only available instructions (or operations) are those described below: 1. Copy inputs into cells of the execution trace. 2. $\texttt{ADD}$ instruction: Add two values in cells on the same row, and leave the result in the first cell of the next row. 3. $\texttt{TIMES4}$ instruction: Multiply a given value by a constant, and leave the result in the first cell of the next row. 4. $\texttt{MUL}$ instruction: Multiply two values in cells on the same row, and leave the result in the first cell of the next row. Letâ€™s provide a specific example by setting the input x to $x := (1, 2, 5)$. Substituting the input values in the given computation yields: $$ [(x0 +x1)Â·4]Â·x2 =[(1+2)Â·4]Â·5=60 $$ The corresponding execution trace, using only the above-mentioned operations, is as follows: !Figure: Letâ€™s take a step-by-step walk-through the above execution trace: 1. First of all, we use the instruction that copies the inputs 1 and 2 into the columns $\texttt{A}$ and $\texttt{B}$, respectively. And then invoke the $\texttt{ADD}$ instruction. 2. After executing the $\texttt{ADD}$ instruction, the second row of register $\texttt{A}$ now holds the sum of 1 and 2, that is the value 3. At this point we utilize the $\texttt{TIMES4}$ instruction to obtain $3 \cdot 4 = 12$, and place it in the first cell of row 3. 3. Now, with 12 in the third row of register $\texttt{A}$, we proceed to copy the third input, 5, into register $\texttt{B}$. Then the $\texttt{MUL}$ instruction is invoked to multiply the current two values in row 3. (viz. 12 and 5). The resulting outcome (which is 60) of the $\texttt{MUL}$ instruction is placed in the first cell of row 4. 4. Consequently, this last outcome 60 is in fact the final output for the entire computation. Observe that we have only used the available instructions as proposed in our scenario. Notice that for a specific computation, and except for the entries in the matrix, the shape of the execution matrix remains the same irrespective of the variable input values $x = (x0, x1, x2)$. For example, if $x = (5, 3, 2)$ for the same program, the resulting execution matrix is as follows: !Figure: However, if we examine the columns in the above execution trace, we can identify two distinct types of columns: - Witness columns: Values in these columns depend on the input values. They change whenever different input values are used. In the above example, columns $\texttt{A}$ and $\texttt{B}$ are witness columns. - Fixed columns: Values in these columns remain unaffected by changes in the input values. In the above example, column $\texttt{C}$ is a fixed column, and it is only used to store the constant 4. In each case of the above example, the constant 4 is used in the second step of the computation, when the $\texttt{TIMES4}$ instruction is invoked. # json-rpc-to-proof.md: This document describes the interaction of users with the Polygon zkEVM network through a component known as zkEVM JSON-RPC. It provides, - An overview of how L2 blocks are generated. - A detailed explanation of how transactions sent through JSON-RPC are processed, including their validation before being stored in the PoolDB. The main components involved in this process are the JSON-RPC, PoolDB, sequencer, executor, prover, HashDB, and Node StateDB. zkEVM JSON-RPC The Ethereum JSON-RPC is a standardized collection of methods used by all Ethereum nodes, acting as the main interface for users to interact with the network. The zkEVM JSON-RPC is an interface compatible with the Ethereum JSON-RPC for any layer 2 system. It implements the same endpoints as the Ethereum JSON-RPC, and similarly receives queries and provides answers to users. The figure below illustrates a user's Ethereum JSON-RPC query to obtain information on the latest forged block. !Figure: User queries Ethereum JSON-RPC for last block The userâ€™s retrieved data comprises all information related to the block, including its transactions. Although the zkEVM JSON-RPC incorporates all Ethereum JSON-RPC methods, some of its answers to queries are only relevant to the zkEVM context. The user can request for the latest block as shown below: !Figure: How user queries The zkEVM JSON-RPC not only includes methods from the Ethereum JSON-RPC, but it also offers additional methods to handle zkEVM-specific functionalities. The complete list of available endpoints and their current implementation status can be obtained in the json-rpc-endpoints.md. The API specification for the zkEVM endpoints, which follows the OpenRPC standard, is available in the endpointszkevm.openrpc.json file. Ethereum JSON-RPC responds with the most recently created L1 block when requesting the last block. zkEVM batches transition through various states: trusted, virtual, and consolidated. Next, we explore the expected format of data in the zkEVM JSON-RPC's responses to queries for the latest L2 block. L2 blocks and batches In general, it is important to establish a clear distinction between batches and blocks. Within the zkEVM framework, an L2 batch represents the minimal data unit for which a proof is generated. On the other hand, an L2 block is the data form returned to users through the RPC, and each block belongs to some L2 batch. L2 blocks, similar to their associated batches, go through all three zkEVM states: trusted, virtual, and consolidated. And, from the userâ€™s perspective, the visible entity is the L2 block. The primary consideration when defining an L2 block is to minimize any delay in batch processing. Based on this premise, L2 users can decide on their acceptable latency by choosing among the three L2 states: trusted, virtual, or consolidated. Clearly, settling for L2 blocks while their associated batches are in the trusted state provides minimal latency and thus the best choice. The delay in this case is simply the $\mathtt{close\a\batch\time}$. Therefore, to ensure reduced delay, we must generate at least one L2 block completely before its associated batch is closed. Dragonfruit upgrade (ForkID 5) Generating L2 blocks means incorporating transactions as soon as they are ordered. That is, a transaction is included in a block once it is established that it will form part of a batch at a specific location. And of course, the fastest scenario is to create an L2 block that consists of only one ordered L2 transaction. This minimizes the delay because each ordered transaction is processed immediately, and without the need to wait for additional transactions before the block is closed. The delay that occurs with each block generation is equivalent and referred to as the $\mathtt{order\a\transaction\time}$. All Polygon zkEVM versions up to the Dragonfruit upgrade (which is the ForkID 5) adopted Optimism's approach to block definition: One L2 block consisting of only one L2 transaction. The formation of a block therefore occurs when the sequencer decides to include a transaction in a batch, that is in the trusted state. Consequently, every batch contains as many blocks as transactions. Also, batch data includes only one timestamp which is shared across all the blocks within the batch. The figure below illustrates the batch structure. !Figure: Etrog upgrade (ForkID 6) The implementation approach of the Dragonfruit upgrade and prior zkEVM versions assigns one ordered transaction to a block, reducing delay to a minimum threshold of the $\mathtt{order\a\transaction\time}$. The Dragonfruit upgrade approach has some drawbacks: - Bloated databases: Since there are as many blocks as there are transactions in each batch, it results in a significant amount of data in the database. - Incompatibility with dApps: It cannot offer a method for assigning a unique timestamp to each block within a batch, causing breaks in Dapps that rely on this parameter to determine the timing of actions performed by smart contracts. The Etrog upgrade (which is ForkID 6) addresses these two issues. That is, each L2 block consists of one or more transactions, and each block has its own unique timestamp. With this implementation approach, the sequencer needs to ensure that block construction has a shorter timeout than the batch generation. The sequencer can modify the timestamp of the various blocks in a batch by utilizing a specific transaction as a marker, dubbed changeL2Block. This transaction is included in the batch to indicate a change from one block to the next, is responsible for modifying the timestamp and the L2 block number. The figure below depicts the structure of the Etrog upgrade's batches. !Figure: Fork-ID 6 - batch approach Read the Etrog upgrade document for further details and how it differs from the Dragonfruit upgrade. Custom zkEVM endpoints When a user requests information from the JSON-RPC regarding the latest block, the response shows the most recent L2 block. The L2 block contains the most recent transactions approved by the trusted sequencer. The information is retrieved with the same RPC call as in Ethereum. Users are less concerned with the transaction being in a block, and more with when the newly generated batch results in a state transition. The zkEVM protocol has additional endpoints for retrieving various information pertaining to the state of the L2 block. For example, the user may query whether a block is in the virtual state or not by using a specific endpoint as shown in the figure below. !Figure: Checking if block is virtualized Here is a list of custom zkEVM endpoints, each accompanied by a brief description: - $\mathtt{zkevm\consolidatedBlockNumber}$: Returns the latest block number within the last verified batch. - $\mathtt{zkevm\isBlockVirtualized}$: Returns true if the provided block number is in a virtualized batch. - $\mathtt{zkevm\isBlockConsolidated}$: Returns true if the provided block number is in a consolidated batch. - $\mathtt{zkevm\batchNumber}$: Returns the latest batch number. - $\mathtt{zkevm\virtualBatchNumber}$: Returns the batch number of the latest virtual batch. - $\mathtt{zkevm\verifiedBatchNumber}$: Returns the batch number of the latest verified batch. - $\mathtt{zkevm\batchNumberByBlockNumber}$: Returns the batch number of the batch containing the given block. - $\mathtt{zkevm\getBatchByNumber}$: Fetches the available info for a batch based on the specified batch number. Sending raw transactions To submit a transaction to the PoolDB, a user invokes the $\mathtt{eth\sendRawTransaction}$ endpoint. When an L2 transaction is received, the JSON-RPC node sends it to the pool component. The pool component is, among other things, responsible for adding transactions into the PoolDB. It therefore conducts initial validations on transactions. If any of these validations fail, an error is sent to the JSON-RPC component, which then forwards it to the user. For valid transactions, the pool conducts pre-execution in which it estimates transaction costs by using the current state root, which may be different when the transaction is ultimately ordered. As illustrated in the figure below, validation is the first step that takes place in the pool, invoking the validateTx() function. The figure below depicts the submission process. !Figure: Sending txs to the system The validateTx() function performs the following preliminary checks: 1. The transaction IP address has a valid format. 2. The transaction fields are properly signed (in both current and pre-EIP-155). EIP-155 states that we must include the chainID in the hash of the data to be signed (which is an anti-replay attack protection). 3. The transactionâ€™s chainID is the same as the poolâ€™s chainID (which is the chainID of the L2 Network) whenever chainID is not zero. 4. The transaction string has an encoding that is accepted by the zkEVM. (See the next subsection for more details on this encoding.) 5. The transaction senderâ€™s address can be correctly retrieved from the transaction, using the ecRecover algorithm. 6. The transaction size is not too large (more specifically, larger than 100132 bytes), to prevent DoS attacks. 7. The transactionâ€™s value is not negative (which can be the case since we are passing parameters over an API). 8. The transaction senderâ€™s address is not blacklisted. That is, it is not blocked by the zkEVM network. 9. The transaction preserves the nonce ordering of the account. 10. The transaction sender account has not exceeded the allowed maximum number of transactions per user to be contained in the pool. 11. The pool is not full (currently, the maximum number of elements in the queue of pending transactions is 1024). 12. The gas price is not lower than the set minimum gas price. This is explained in more detail in the Effective gas price section. 13. The transaction sender account has enough funds to cover the costs $$ \mathtt{value} + \mathtt{GasPrice} \cdot \mathtt{GasLimit} $$ 14. The computed intrinsic gas is greater than the provided gas. The intrinsic gas of a transaction measures the required gas in terms of the amount of transactional data plus the starting gas for the raw transaction which is currently of 21000, or 53000 in the case of a contract creation transaction. 15. The current transaction GasPrice is higher than the other transactions in the PoolDB with the same nonce and from address. This is because a transaction cannot be replaced with another with lower GasPrice. 16. The sizes of the transactionâ€™s fields are compatible with the Executor needs. More specifically: - Data size: 30000 bytes. - GasLimit, GasPrice and Value: 256 bits. - Nonce and chainID: 64 bits. - To: 160 bits. zkEVM customized transaction encoding As mentioned in the fourth item of the above list of validation checks, this subsection pertains to the zkEVM's custom encoding of transactions using the $\texttt{validateTx()}$ function. Currently, the zkEVM only supports non-typed transactions. You can refer to standard transaction encodings in Ethereum for further information. The $\texttt{to-be-signed-hash}$ remains consistent with pre-EIP155 and EIP155 transactions, resulting in the respective signatures for these transactions: $$ \begin{aligned} &\texttt{to-be-signed-hash}{\texttt{pre-EIP155}} = \mathtt{keccak(rlp(nonce,gasPrice,startGas,to,value,data))}\\ &\texttt{to-be-signed-hash}{\texttt{EIP155}} = \texttt{keccak(rlp(nonce,gasPrice,startGas,to,value,data,chainID,0,0))}\\ \end{aligned} $$ The transaction string used for L2 transactions included in the batches is a slightly modified version of the regular transaction string. The reason for this is to streamline the proving systemâ€™s processing of L2 transactions. Analyzing typical transactions Letâ€™s analyze how an EIP-155 transaction is handled in a chain with $\texttt{chainID = 1101}$ and $\texttt{parity = 1}$. Typically, the following transaction string is received: $$ \texttt{rlp(nonce,gasPrice,startGas,to,value,data,r,s,v = 2238)} $$ To check the signature, the received string is rlp-decoded in order to extract the $\texttt{nonce}$, $\texttt{gaspPrice}$, $\texttt{startGas}$, $\texttt{to}$, $\texttt{value}$, $\texttt{data}$, $\texttt{r}$, $\texttt{s}$ and $\texttt{v}$. This is followed by computing the chain identifier and the parity from the given $\texttt{v}$ value: $$ \begin{aligned} &\texttt{chainID} = âŒˆ(\texttt{v} âˆ' 35)/2âŒ‰ = âŒˆ(2238 âˆ' 35)/2âŒ‰ = 1101.\\ &\texttt{parity} = \texttt{v} âˆ' \texttt{(2 Â· chainID)} âˆ' 35 = 2238 âˆ' (2 Â· 1101) âˆ' 35 = 1. \end{aligned} $$ Next, compute the rlp-encoding of the following parameters: $$ \texttt{rlp}(\texttt{nonce}, \texttt{gasPrice}, \texttt{startGas}, \texttt{to}, \texttt{value}, \texttt{data}, 1101, 0, 0) $$ and compute its Keccak hash: $$ \texttt{to-be-signed-hash} = \texttt{keccak}\big(\texttt{rlp}(\texttt{nonce},\texttt{gasPrice},\texttt{startGas},\texttt{to},\texttt{value},\texttt{data}, 1101, 0, 0)\big) $$ Finally, validate the ECDSA signature over $\texttt{to-be-signed-hash}$ with $\texttt{r}$, $\texttt{s}$ and the parity. As seen above, the rlp-decoding and rlp-encoding are performed on the transaction parameters. After creating the transaction string as the $\texttt{to-be-signed-hash}$, concatenate the $\texttt{r}$, $\texttt{s}$ and parity, which are the only extra parameters required to verify the signature. According to the above logic, use the following raw transaction string for the zkEVM L2 transactions: $$ \texttt{rlp}(\texttt{nonce},\texttt{gasPrice},\texttt{gasLimit},\texttt{to}, \texttt{value},\texttt{data}, \texttt{chainID}, 0, 0)\| \texttt{r} \| \texttt{s} \| \texttt{v} \| \texttt{effectivePercentage} $$ The $\texttt{parity}$ is expressed in the one-byte $\texttt{v}$ parameter that can take the pre-EIP155 $\texttt{parity}$ values: 27 and 28. The parameter $\texttt{effectivePercentage}$ is a zkEVM-specific parameter related to the gas fee system. Since the transaction signature is the same, it is easy to transform standard transactions into the zkEVM's custom format and vice-versa. Find JS utilities for carrying out these transformations in the utilities. Sequencer batch pre-execution Once the first checks are completed without any errors, the function $\texttt{StoreTx()}$ is executed. The function performs a transaction estimation pre-execution, which involves estimating zkCounters for pricing and prioritization purposes, though based on a potentially incorrect state root. If an out-of-counters (OCC) error occurs, the transaction is not added to the PoolDB but rather discarded. Further details on this are discussed in the effective gas fee document. Otherwise, a successful transaction estimation pre-execution results in the transactions being stored in the PoolDB. !Figure: Second batch pre-execution The sequencer chooses transactions from the PoolDB to include in blocks before batch-closure, then inserts all included transactions into a designated table within the nodeâ€™s StateDB. The sequencer also updates the HashDB with the L2 state data obtained by pre-executing the batch again. This batch pre-execution performed by the sequencer is actually the second one. At this stage of transaction processing, the batch pre-execution is expected to be correct because the sequencer uses the current root to calculate the exact gas costs and required zkCounters. The figure above illustrates the process. After the above process, the executor performs the last execution known as batch execution during which the prover can generate the corresponding proof. If all the above-mentioned computations are correct, the second pre-execution and the final execution should be identical. The whole process is summarized in the figure below. It depicts three executions occurring between when the user sends a transaction via the JSON-RPC and the generation of a proof. !Figure: Three execution between sending tx and proof generation Note that the counters check is performed twice, even as depicted in the figure above. The first check is an estimation, while the second one provides the actual values. As previously mentioned, the two final checks should yield the same outcome if all computations are correct. # l2-state-tree.md: This document explains how the zkEVM proving system manages changes in the L2 state tree. The main components of the zkEVM proving system are, the ROM, the prover HashDB, and the storage state machine's executor. In the zkEVM context, as in Ethereum, it is crucial to ensure the accuracy of state changes resulting from the execution of transactions. zkEVM ROM The zkEVM ROM, written in zero-knowledge assembly (zkASM), is a program designed to prove computations related to correct L2 state transitions resulting from L2 transactions in a given batch. Often, multiple zkASM instructions are used to implement a single zkEVM opcode. The L2 state is captured in the form of a Merkle tree, and its Merkle root, referred to as the state root, uniquely represents the summary of the current state data. The zkEVM ROM must, therefore, have the capability to correctly execute the CREATE, READ, UPDATE, and DELETE (CRUD) operations on the Merkle tree. The storage state machine The zkEVM implements a secondary state machine, known as the storage state machine, specifically for generating an execution trace. The execution trace provides evidence for any creation, reading, updating, or deletion of L2 state data. Any operation applied to the Merkle tree must be evidenced with proof, attesting that the tree modification was executed correctly. Such a proof consists of the sibling node and other relevant hash nodes sufficient for verification purposes. This is called a Merkle proof. An example of a Merkle proof can be found in the Concepts section. Verifying a Merkle proof involves using the given information to compute the Merkle root and checking if it matches the actual Merkle root. After processing the last L2 transaction in a batch, the resulting root becomes the new state root. Example. (Proving READ and UPDATE operations) We provide illustrative examples showcasing two CRUD operations on the L2 state: read and update operations. The storage state machine is responsible for providing validity proofs of these operations. !Figure: State change Consider the figure above, where the objective is to read the information stored in the fourth leaf ($l4$). Suppose a user carries out a transaction and $l4$ represents the balance of the account that initiates the transaction. In the process of reading this leaf, it is crucial to verify its inclusion in the current state at the specified position. This is deterministically computed from some unique properties of the stored data, such as the account address or the data type. In the figure above, the blue boxes represent data that must be supplied to the storage state machine for generating the execution trace. As typically required when verifying a Merkle proof, the data stored in the blue boxes enables the storage state machine to validate that the computed root matches the state root of the Merkle tree. !Figure: Consider the figure above, which depicts a scenario where a leaf is updated. A leaf update means changing the leaf's value, which consequently modifies the state root. Consequently, the yellow boxes in the figure represent all affected nodes, as they incorporate information from the updated leaf $l2$. The storage state machine is used once again to generate an execution trace, ensuring the accuracy of all tree modifications. Following the same procedure as in the read case, the consistency of the state root is verified to confirm that the changes have been accurately executed. The storage state machine's executor and HashDB All essential hashes required for computing execution traces are stored in the HashDB. During the execution of the L2 stateâ€™s related zkEVM opcodes, such as SSTORE or SLOAD, the storage state machineâ€™s executor consistently retrieves values from the HashDB and uses them to construct the corresponding execution traces. The HashDB contains all the hashes, which are all the nodes of the Merkle tree of the L2 state. The figure below depicts an interaction of the executor with the HashDB, when proving the state transition from $S^{L2x}i$ to $S^{L2x}{i+1}$. The executor takes as inputs, the current state $S^{L2x}i$ and a batch of transactions, retrieves Merkle proofs from the HashDB, and produces the execution trace representing all changes to the state tree. !Figure: It is important to note that the Merkle tree used in the Polygon zkEVM is a binary sparse Merkle tree (SMT), distinguishing itself from the Merkle tree used in the L1 EVM. # l2statetree-keys-and-values.md: The Polygon zkEVM state is stored in a key-value format within a special Merkle tree known as a Sparse Merkle Trie (SMT). This tree is binary and fully updatable, supporting both Read and Write operations. Similar to the use of Merkle Patricia tries in Ethereum, an SMT is a data structure designed to retrieve a value by traversing a branch of nodes that store associated references, called keys, which together lead to a leaf containing the value. Underlying Hash Function Since Polygon zkEVM is a zk-rollup, a ZK-friendly hash function is the most suitable for constructing SMTs. The Poseidon hash function, designed to address some of the challenges and limitations of traditional hash functions in implementing ZK-proofs, was a natural choice. Readers interested in cryptographic details should refer to the sponge construction document for Poseidon's basic design structure and the Poseidon state machine document for its internal mechanism. We henceforth consider an instantiation of Poseidon over the Goldilocks field $\mathbb{F}p$ where $p = 2^{64} âˆ' 2^{32} + 1$. Poseidon inputs and outputs Observe that Poseidon's input and output elements are represented as arrays. The Poseidon hash function accepts a full input consisting of 12 field elements: an array of 4 capacity elements and an array of 8 input elements, with each input element being a 32-bit field element. The output is an array of 4 field elements, each approximately 64 bits long. Refer to the figure below for a simplified diagram of Poseidon's input and output arrays. !Figure: Poseidon input output Poseidon hash types To safeguard SMTs against second-preimage attacks, two instances of the Poseidon hash function are utilized: $$ \mathtt{hashType = 0}\ \text{ and }\ \mathtt{hashType = 1} $$ Referring to the figure above, the hashType is indicated by the first element, $\mathtt{capacity[0]}$, of one of the following two arrays: $$ \mathtt{capacity = [0, cap1, cap2, cap3]} \text{ or } \mathtt{[1, cap1, cap2, cap3]}. $$ That is, $\mathtt{hashType = capacity[0] \in \{ 0, 1 \}}$. When constructing an SMT, the Poseidon hash with $\mathtt{hashType = 0}$ is used to create branch nodes, while $\mathtt{hashType = 1}$ is used for leaf nodes. Example (hashType) Denote a Poseidon hash as follows: $$ \texttt{Poseidon(capacity; values)} $$ where $\mathtt{capacity}$ is an array of 4 field elements and $\mathtt{values}$ is an array of 8 input elements of the hash. Using this notation, the two instances of the Poseidon hash function are: - $\mathtt{Poseidon(0; \cdot)}$ corresponding to $\mathtt{hashType = 0}$, and used for branch nodes. - $\mathtt{Poseidon(1; \cdot)}$ corresponding to $\mathtt{hashType = 1}$, and used for leaf nodes. Constructing an SMT Letâ€™s use an example to visualize how an SMT is constructed. Suppose a given SMT has values stored at leaves associated with the following four keys: $$ \mathtt{101101},\ \mathtt{000110},\ \mathtt{001011} \texttt{ and } \mathtt{001100}. $$ Firstly, note that by non-zero leaves we refer to leaves storing some data. Hence, a zero leaf refers to a leaf node that carries no data. Secondly, when constructing the SMT corresponding to these 4 keys, the key-bit $\texttt{0}$ represents an edge to the left, while the key-bit $\texttt{1}$ means an edge to the right. This means, non-zero leaf nodes share paths for as long as their associated keys have common least-significant bits. Therefore, any pair of keys associated with distinct non-zero leaves must differ in at least one of the key bits. This means that prefixes (short bit strings formed from consecutive least-significant key bits) can be used to accurately position non-zero leaf nodes. Now, based on the keys provided above, there are four distinct prefixes: $$ \texttt{1},\ \texttt{000},\ \texttt{0010} \texttt{ and } \texttt{0011}. $$ Observe that: 1. The path to the leaf associated with $\texttt{key} = \mathtt{101101}$ differs from the other three paths in the very first edge from the root, which is the edge to the right. Assuming there are no other non-zero leaf nodes, the corresponding leaf is therefore positioned at the first child node of the root, along the edge to the right. 2. Since the prefixes $\texttt{000},\ \texttt{0010} \texttt{ and } \texttt{0011}$ share the first two bits $\texttt{00}$, the paths to the associated leaves coincide only for the first two edges from the root, both of which are to the left. 3. The prefix $\texttt{000}$ differs from the other two prefixes ($\texttt{0010}$ and $\texttt{0011}$) at the third bit. Hence, assuming there are no other non-zero leaves, the leaf associated with $\texttt{key} = \mathtt{000110}$ is positioned at the third child node, at the end of the third edge from the root, following three consecutive left edges. 4. The last two prefixes ($\texttt{0010}$ and $\texttt{0011}$): Share a common third key bit 1, so the paths to their associated leaves coincide up to the third edge, which is to the right. Only differ at their fourth key bits, meaning the fourth edge of the path to the leaf associated with $\mathtt{key = 001011}$ is to the left, while the fourth edge of the path to the other leaf is to the right. Again, assuming there are no other non-zero leaves in this SMT, each these two non-zero leaf nodes (associated with prefixes $\texttt{0010}$ and $\texttt{0011}$) is positioned at the end of the fourth edge from the root. The figure below depicts the SMT containing the four non-zero leaves associated with the given keys: $\mathtt{101101}$, $\mathtt{000110}$, $\mathtt{001011}$ and $\mathtt{001100}$. !Figure: SMT example Node types As depicted in the figure above, a typical SMT has four types of nodes: leaf, root, branch, and zero nodes. 1. The leaf node encapsulates the actual data associated with a key. The data is captured in the form of a hash digest of the key-value pair. And, due to Poseidon's cryptographic properties such as collision-resistance, anyone can use a Merkle proof to verify if indeed certain data is stored in a particular leaf of the SMT. 2. The root node is the highest node in an SMT, the sole node at level zero, connecting all branches and leaves. It's the starting point for traversing down the tree structure. As a hash that includes hashes of all data captured in the SMT, the root serves as the cryptographic summary of the entire dataset and thus acts as a fingerprint of all the values stored in the SMT. 3. A branch node refers to any intermediary node that lies between the root and leaf nodes, between the root node and other branch nodes, or between a branch node and leaf nodes. Branch nodes are therefore positioned at various levels in an SMT, and store hashes derived from concatenating two hashes of their child nodes. The hash at each branch node cryptographically encapsulates the data contained in the subtree (all its child nodes, grandchild nodes, including the leaf nodes) that has the branch node as its root node. 4. A zero node does not represent a zero hash value associated with a specific key, but instead denotes a branch node with no subtree beneath it. These type of nodes exist whenever two conditions are satisfied: - No key associated with non-zero values stored in the SMT has a prefix that leads to the node. - The node has a sibling node, which is either a non-zero leaf or a branch node with a non-zero leaf as its child or grandchild. Referring to the above example, the node reached by traversing the 2-edge path $\mathtt{01}$ is a zero-node because: Firstly, none of the keys ($\mathtt{101101}$, $\mathtt{000110}$, $\mathtt{001011}$ and $\mathtt{001100}$) has $\mathtt{01}$ as a prefix. Secondly, the sibling node reached by traversing the 2-edge path $\mathtt{00}$ has 3 non-zero grandchild nodes in its subtree. Keys and values The aim with this subsection is to provide an understanding on how the L2 state is managed in terms of the key-value data within SMTs. As seen above, keys are crucial in positioning values, as well as traversing SMTs to locate these values. Leaf nodes serve as the endpoints of the tree and essentially hold data. Next, we provide details on the derivation of keys and the computation of values. Deriving keys We begin by describing how keys are derived from their associated leaf data. Node key is computed as follows: $$ \mathtt{leafKey = Poseidon(capacityHash; addr, 0, SMTKEY, 0)} $$ Each of the four arguments is described below: 1. The address $\mathtt{addr}$ consists of 5 32-bit segments representing the Ethereum address associated with the leaf, having 160 bits in the EVM: $$ \mathtt{addr = (addr[0], addr[1], addr[2], addr[3], addr[4])} $$ 2. The SMT key is a single field element, $\mathtt{SMT{KEY} \in \{ 0, 1, 2, 3, 4 \}}$, indicating which of the following five L2 state information is being stored: - $\mathtt{SMTKEY = 0}$: for the account balances. - $\mathtt{SMTKEY = 1}$: for the account nonces. - $\mathtt{SMTKEY = 2}$: for the smart contract code. - $\mathtt{SMTKEY = 3}$: for the smart contract storage. - $\mathtt{SMTKEY = 4}$: for the smart contract length. 3. The hash of the capacity array, called $\mathtt{capacityHash}$, is computed in one of two ways, depending on whether the stored data is a storage slot or some other type of data. (a) When storing data that is not a storage slot, $\mathtt{capacityHash}$ is computed as follows: $$ \mathtt{capacityHash = Poseidon(0; 0, 0, 0, 0, 0, 0, 0, 0)}, $$ where capacity is shown as $\texttt{0}$, but it's in fact an array of 4 field elements, each set to $\texttt{0}$. Since a single smart contractâ€™s address can have multiple storage slots, a key $\texttt{storageKey}$ is created for each slot. (b) For each storage slot, $\mathtt{capacityHash}$ is computed with a specific $\texttt{storageKey}$ as follows: $$ \mathtt{capacityHash = Poseidon(0; storageKey)} $$ where $\texttt{storageKey}$ is an array of 8 field elements each constrained to 32-bit values representing storage slot identifiers, which in the EVM have a size of 256-bits. Computing values We now explain how the values stored within leaf nodes are obtained. The current methodology for hashing the leaf data to derive the leaf nodeâ€™s value is as follows: $$ \mathtt{leafValue = Poseidon(1; remLeafKey, valueHash)}. $$ The first argument is the capacity set to $\texttt{1}$, which is actually an array of 4 field elements: $\mathtt{capacity = [1,0,0,0]}$. The second argument, $\texttt{remLeafKey}$, is an array of 4 field elements representing the remaining key associated with the leaf node. The third argument, $\texttt{valueHash}$, is an array of 4 field elements calculated as follows: $$ \mathtt{valueHash = Poseidon(0; value)}, $$ where $\texttt{value}$ is an array of 8 field elements, each restricted to a 32-bit length. The 32-bits values in the array are computed differently, depending on the type of the data stored in the tree. 1. For $\texttt{balances}$, $\texttt{nonces}$ and $\texttt{storageSlotValues}$. In the EVM, $\texttt{balances}$, $\texttt{nonces}$ and $\texttt{storageSlotValues}$ are of 256 bits. So, their $\texttt{valueHashes}$ are computed as expected: $$ \begin{aligned} &\texttt{valueHash = Poseidon(0; balance)},\\ &\texttt{valueHash = Poseidon(0; nonce)},\\ &\texttt{valueHash = Poseidon(0; storageSlotValue)}. \end{aligned} $$ where each input ($\texttt{balances}$, $\texttt{nonces}$ or $\texttt{storageSlotValues}$) is an 8 field element array, each constrained to 32-bit values. 2. For smart contract bytecode. The hash of the smart contract bytecode is also stored in the L2 state tree. However, since bytecode has an arbitrary size, it can exceed 256-bits (32 bytes). To manage this issue: Firstly, padding is used to adjust the size of the input bytecode to a suitable multiple, in particular, a multiple of 56 bytes. Secondly, the resulting padded byte string is linearly hashed. !Figure: bytecode and padding 3. For bytecode length. The length of the actual bytecode is stored in the L2 state tree, because it is necessary to distinguish between payload and padding. The bytecode length which is the number of bytes forming the bytecode, is hashed as follows: $$ \texttt{valueHash = Poseidon(0;bytecodeLength)}. $$ 4. Input bytecode encoding. We encode the bytecode in segments of 7 bytes (56 bits), which is the biggest amount of bytes that can be represented with an element of the Goldilocks field: $$ 2^{56} < 2^{64} â^'2^{32} +1 < 2^{64}. $$ Since the input values of the Poseidon hash function is an array of 8 field elements, the total input is encoded as $\mathtt{ 7\ bytes\ Ã—\ 8\ elements = 56\ bytes }$ of bytecode. 5. Padding procedure. The padding procedure used to adjust the original bytecode string to a padded string, that's a multiple of 56 bytes, is as follows: - Add the byte $\texttt{0x01}$ at the end of the original bytecode. - Fill with zeros until the padded $\mathtt{bytecode\ length + 1 }$ is multiple of 56 bytes. - Add $\texttt{0x80}$ as the last byte. 6. Example (Bytecode hashing) Suppose the 2-byte, $\mathtt{0xdead}$, is the original bytecode. The padded bytecode becomes: $$ \texttt{0xdead01000000000000000000...000000000000080} $$ a string of 56 bytes. 7. Recursively chained hashing. Smart contract bytecodes larger than 256 bits undergo a chained hashing process to obtain a valid representative value (called $\texttt{bytecodeHash}$). More specifically, the padded bytecode is split into chunks of 7 bytes, and 8 of these chunks are recursively added as input values of the Poseidon hash function. Although the initial capacity value is $\mathtt{capacity = [0,0,0,0]}$, all subsequent capacity values are to the previous output of the Poseidon hash function. !Figure: Poseidon hash repeated 8. Example. (Recursively chained Poseidon) To illustrate this, let's follow on with the previous 2-byte example, $\texttt{0xdead}$, which was padded to: $$ \mathtt{0xdead010000000000000000000000000...000000000000000000000080} $$ Since the 56-byte padded bytecode amounts to 8 chunks of 7 bytes, performing a single Poseidon hashing is sufficient. By setting the capacity to an array $\mathtt{capacity = [0,0,0,0]}$, where each $\texttt{0}$ is a 7-byte zero, the total input to the Poseidon hash function is as below: !Figure: Poseidon hash - 12 elements ... 7 bytes values Finally, the $\texttt{valueHash}$ of the bytecode is $\texttt{valueHash = Poseidon(0; bytecodeHash)}$. Conclusion The above construction of SMTs is efficient and optimal in many ways. For instance, the design approach to avoid hashing the full height of an empty branch, but rather store it as a zero-leaf where the value stored is an explicit zero value, achieves huge storage savings. In practical terms, this means that the branch node at level 1 assumes a specific value: $$ \mathtt{Poseidon(0; leftChildNode, 0). } $$ This type of optimization is called partial tree construction, where leaf nodes are at various levels in the tree. # order-and-prove.md: Order and prove In this document, we distinguish between the sequencing and proving processes, with emphasis on the need to distinguish between the two stages of the overall transaction flow. The decoupling of sequencing from proving enhances the system's efficiency. Central to the zkEVM architecture are the verifier smart contract, the prover, the aggregator, and sequencer. Typical state transition Recall that, a proof of correct execution of transactions in a batch is generated, and the proof needs to be verified by the verifier smart contract. A new L2 state is reached, and it is dubbed the consolidated state. The processed batch is referred to as a 'consolidated batch'. That is, it has been verified by the L1 smart contract. Submitting the proof πâ€‹, together with the corresponding publics, for verification of correct execution, is an L1 transaction. Once the smart contract successfully verifies this L1 transaction, this proves that the L2 state has correctly evolved from the old state $S^{L2x}i$ to a new state $S^{L2x}{i+1}$ according to the processed batch. The figure below depicts a state transition. !Figure: zkEVM key performance indicators The zkEVM's key performance indicators (KPIs) are Delay and Throughput. Delay refers to the time elapsed from when a user sends an L2 transaction until the transaction's execution results are reflected in the L2 state. This is a major KPI when it comes to positive user experience (UX). Throughput measures the system's capacity of processing transactions. It can be computed in transactions per second, gas per second, or batches per second. Let's analyze and possibly re-engineer the current system to improve these KPIs. There are three parameters affecting these KPIs: $\mathtt{close\a\batch\time}$, $\mathtt{prove\a\batch\time}$, and $\mathtt{block\time}$. 1. $\mathtt{close\a\batch\time}$: The time taken to get sufficient transactions to close a batch or a given timeout for this purpose. !Figure: closeabatchtime 2. $\mathtt{prove\a\batch\time}$: The time taken to generate a proof for a single batch. The size of the batch can obviously affect this time. !Figure: proveabatchtime 3. $\mathtt{block\time}$: This is the minimum time taken to execute L1 transactions. !Figure: closeabatchtime Let's explore how this parameters impact the Delay and Throughput of the full system. Processing pipeline Consider an example of a simplified processing pipeline, similar to an assembly line in a factory, as illustrated in the figure below. Identify two key performance indicators (KPIs) of interest: lead time (or delay) and production rate (or throughput). - Lead time refers to how long it takes to produce the first car after starting the line. The lead time, as indicated in the figure above, is 6 hours. - Production rate tells us how many cars can be produced per unit of time. The production rate is 1 car every 6 hours, which is equivalent to $\mathtt{\frac{1}{6}}$ cars per hour. !Figure Two scaling methods The following question arises: How can we improve both delay and throughput metrics? The objective is to increase the throughput and reduce the delay. Two methods are employed: horizontal scaling and vertical scaling. - Horizontal scaling involves adding more operators of smaller power in parallel, boosting throughput. In which case, the delay remains the same. (In our system, operators are equivalent to CPUs.) - Vertical scaling, on the other hand, entails adding more powerful operators. This is equivalent to adding faster CPUs. In this case, delay is reduced but throughput is increased. As expected, vertical scaling is more expensive to implement compared to horizontal scaling. Consider the figure below, depicting two scenarios of scaling on the previous processing pipeline scenario. (a) Serial horizontal scaling. Replacing the 3-hour operator (at the engine-mounting stage) with three 1-hour operators working serially. And, replacing the 2-hour operator (at the body-mounting stage) with two 1-hour operators. (b) Vertical scaling. Replacing the 3-hour operator (at the engine-mounting stage) with one 3-times faster operator, capable of performing the normal 3-hour engine-mounting task in only one hour. And, replacing the 2-hour operator (at the body-mounting stage) with one 2-times faster operator, capable of performing the usual 2-hour body-mounting task in just an hour. !Figure: serial CPUs vs. faster CPU Initially, before any scaling was applied, the delay was 6 hours and the throughput was $\mathtt{\frac{1}{6}}$â€‹â€‹ cars per hour. The serial horizontal scaling, (a), does not result in any improvement on the delay and throughput. But the vertical scaling, (b), results in the delay of 3 hours, while the throughput increases to 1 car every 3 hours, which is equivalent to $\frac{1}{3}$ cars per hour. Improving batch processing KPIs In the current design, batch processing consists of: closing a batch, generating the proof, and submitting the verify-this-batch L1 transaction to the verifier smart contract. The two zkEVM KPIs can be computed as follows: - Delay is computed with this formula, $$ \texttt{delay} = \mathtt{close\a\batch\time} + \mathtt{prove\a\batch\time} + \mathtt{block\time} [\text{seconds}] $$ - And throughput is given by this formula, $$ \texttt{throughput} = \dfrac{1}{\mathtt{prove\a\batch\time}}\ [\text{batches per second}] $$ When computing throughput, we assume that closing, proving, and verifying a batch can be done in parallel with other batches. And thus, in practice, throughput is determined by the longest part of the process, which is to prove the batch. $$ \begin{aligned} &\mathtt{close\a\batch\time} << \mathtt{prove\a\batch\time} \\ &\mathtt{block\time} << \mathtt{prove\a\batch\time} \end{aligned} $$ To provide specific numbers: $$ \begin{aligned} &\mathtt{block\time} = 15\ \text{seconds}\ (\text{average})\\ &\mathtt{prove\a\batch\time} = 120\ \text{seconds}\ (\text{min})\\ &\mathtt{close\a\batch\time} = 3\ \text{seconds}\ (\text{max})\\ \end{aligned} $$ Improving KPIs with vertical scaling The aim is to increase throughput and reduce delay. The one limiting factor in this case is the $\mathtt{prove\a\batch\time}$. Vertical scaling means adding more resources to the existing machines. It can be achieved by running provers on more powerful machines, optimizing the proving system, or a combination of both. Although vertical scaling seems like a straightforward solution to speed up proof generation, it has limitations: - Cost-effectiveness: Upgrading to very powerful machines often results in diminishing returns. The cost increase might not be proportional to the performance gain, especially for high-end hardware. - Optimization challenges: Optimizing the prover system itself can be complex and time-consuming. Improving KPIs with horizontal scaling Another option is to scale the system horizontally. Horizontal scaling involves adding more processing units (workers) to distribute the workload across multiple machines and leverage additional hardware resources in parallel. In the context of a batch processing system, this translates to spinning up multiple provers to work in parallel. Naive horizontal scaling Consider the figure below, depicting a naive implementation of horizontal scaling, which involves: 1. Parallelized proof generation by spinning up multiple provers. 2. Proof reception, where each prover individually sends the proof it generated to the aggregator. 3. Proof Verification, which means the aggregator puts all these proofs into an L1 transaction, and sends it to the smart contract for verification of batches. !Figure: Naive approach - Horizontal scaling This approach means closing batches serially, while generating their proofs in parallel. Notice that, as depicted in the figure above, the proofs $\pia$, $\pib$ and $\pic$ are serially input to the L1 smart contract for verification. This means the overall verification cost is proportional to the number of proofs sent to the aggregator. The disadvantage with the naive approach is the associated costs, seen in terms of the space occupied by each proof, and cumulative verification expenses with every additional proof. Proof aggregation in horizontal scaling Another option is to scale the system horizontally with proof aggregation, as shown in Figure 6. Hereâ€™s how it works: 1. Parallelized proof generation, by instantiating multiple provers. 2. Proof reception, where each prover individually sends the proof it generated to the aggregator. 3. Proof aggregation, where proofs are aggregated into a single proof. 4. Proof verification here means encapsulating only one proof, the aggregated proof, in an L1 transaction. And hence transmitting it to the smart contract for batch verification. The foundation of this approach rests on zkEVM's custom cryptographic backend, designed specifically to support proof aggregation. It allows multiple proofs to be combined into a single verifiable proof. As depicted in the figure below, the proofs $\pia$, $\pib$ and $\pic$ are aggregated into a single proof $\pi{a,b,c,...}$. The key advantage is constant verification costs on L1, regardless of the number of proofs being aggregated. !Figure: Horizontal-scaling proof aggregation Deep dive into horizontal scaling Letâ€™s delve deeper into how the use of proof aggregation boosts the systemâ€™s throughput. A crucial metric in this process is $\mathtt{aggregation\time}$, which represents the time it takes to combine proofs from N batches, which is close to 12 seconds. Throughput, measured in batches per second, can be computed as follows: $$ \dfrac{N}{ \text{max} \big(\mathtt{prove\a\batch\time},\ N Â· \mathtt{close\a\batch\time},\ \mathtt{block\time},\ \mathtt{aggregation\time}\big)} $$ Observe that, since proving and closing batches, and aggregating proofs can run in parallel for a set of N batches, verification of all N batches takes as long as the slowest of the three operations. Hence the denominator, in the above formula, is the maximum among the values: $\mathtt{prove\a\batch\time}$, $N Â· \mathtt{close\a\batch\time}$, $\mathtt{block\time}$, and $\mathtt{aggregation\time}$. This means, in the case where the maximum time in the denominator is $\mathtt{prove\a\batch\time}$, the system's throughput increases by a factor of Nâ€‹. Delay in this scenario can be computed as follows: $$ \texttt{delay} = NÂ·\mathtt{close\a\batch\time} + \mathtt{prove\a\batch\time} + \mathtt{aggregation\time}+ \mathtt{block\time} $$ A straightforward aggregation of batches substantially increases delay relative to a single batch approach. As discussed earlier, delay is a critical factor to user experience. To retain the throughput gains while reducing the delay, we can adopt a two-step approach for batch processing: first, order (also known as sequence) and then prove. This segmentation allows for optimization in each step, potentially reducing the overall delay while maintaining improvements in throughput. Enhancing delay by order then prove The rationale behind decoupling batch ordering (sequencing) from batch proving is twofold: - Ensure swift responses to users regarding their L2 transactions with minimal delay. - Enable transaction aggregation for maximizing system throughput. Sequencing an L2 batch involves deciding which L2 transactions should be part of the next batch. That is, when to create or close the batch, and sent it to L1. As the sequence of batches is written to L1, data availability and immutability are ensured on L1. Sequenced batches may not be proved immediately, but they are guaranteed to be proved eventually. This creates a state within the L2 system that reflects the eventual outcome of executing those transactions, even though the proof hasnâ€™t been completed yet. Such a state is called a virtual state because it represents a future state to be consolidated once the proof is processed. More precisely, the virtual state is the state reached after executing and sequencing batches in L1, before they are validated using proofs. !Figure: Definition of the Virtual state Itâ€™s crucial for users to understand that once a transaction is in the virtual state, its processing is guaranteed. Notable improvement lies in the ability to close batches more rapidly than the block time, providing a more efficient and expedited processing mechanism. Letâ€™s adopt a revised definition for the delay: - The duration from the moment a user submits an L2 transaction until that transaction reaches the virtual state. From the userâ€™s perspective, once the transaction is in the virtual state, it can be regarded as processed. $$ \mathtt{delay}^{(\mathtt{to\virtual})} = \mathtt{close\a\batch\time} + \mathtt{block\time} $$ The smart contract allows us to sequence multiple batches, in which case the delay can be computed as shown below. $$ \mathtt{delay}^{(\mathtt{to\virtual})} = N Â· \mathtt{close\a\batch\time} + \mathtt{block\time} $$ Note that we have experienced a significant reduction in the delay. Below, we present several advantages of decoupling batch sequencing from batch proving: - Queue management: This approach enables effective management of the queue for sequenced batches that await consolidation. - Flexibility in delay and prover resources: It becomes possible to adjust the amount of delay by adjusting the number of provers in operation. - User perception: Decoupling allows for adjustments in delay and resource alloca- tion without impacting the perceived delay experienced by users. # polynom-identity-lang.md: In order to prove correctness of execution, and enable such proofs to be verifiable, a domain-specific language (DSL) called Polynomial identity language (PIL) is employed. The word 'identity' refers to polynomial equations that must be satisfied by values in the execution trace, if all input programs were correctly executed, for the given initial input values. These polynomial identities are equivalent to equations, called arithmetic constraints. that govern correct execution of input programs. These constraints link the execution trace to the input program it models, in a natural way. The next example illustrates this clearly. Example: (Constraints and execution correctness) Consider again our previous example of a computation, $[(x0 +x1)Â·4]Â·x2$, with its corresponding input program: $$ \big(\texttt{ADD},\ \texttt{TIMES4},\ \texttt{MUL} \big) $$ For the given initial input values, $x = (x0, x1, x2) = (1, 2, 5)$, the execution trace and its corresponding constraints are given below. !Figure: Table of program computation and Constraints It can be observed that: - The first two equations check for correct assignment of input values. - The third and fourth equations test output values of $\texttt{ADD}$ and $\texttt{TIMES4}$, respectively. - The fifth equation checks for correct assignment of the third input value, $x2 = 5$. - The last equation tests the output value of $\texttt{MUL}$ Given the set of constraints (on the right) it is easy to examine whether the execution trace (on the left) correctly models the input program for the given initial input values. !!! warning Throughout this document, concepts of the execution trace and the constraints are didactically explained. That is, explanations involve constructing examples, while gradually adding complexity. For this reason, some things like the way the constraints are written is not technically correct, but it will be fixed as we get to the end of the document. Interpolating values in execution matrices Given arithmetic constraints of an execution matrix, its corresponding polynomial identities are derived through a process called interpolation. In simple terms, interpolation means taking 'Cartesian' points expressed as Nâ€‹-tuples, and computing the closest polynomial whose graph passes through the points. The N-tuples are constituted by the values that form each column of the execution matrix. The output of an interpolation is therefore polynomials of degree N, and N is in fact the number of rows of the execution trace (equivalently, the number of values in a column). In the particular case of the zkEVM, interpolation is carried out via Fast Fourier Transformations (FFTs) over a specific domain. That domain is the subgroup of roots of unity, $H = \{ 1H, \omega, \omega^2, \dots , \omega^{2^{k-

1}} \}$ where $\omega^{2^k} = 1H$, the group-identity of H. And, for efficiency of FFTs, the value of N is always set to a power of 2 (i.e., $N = 2^k$ for some natural number $k \in \mathbb{N}$). !Figure: Details of how arithmetic constraints translate into polynomial identities can be found in the concepts section of the documentation here. The polynomial identities uniquely describe the input program modelled by the execution trace. The zkEVM expresses the polynomial identities in the Polynomial identity language (PIL), and stores them in a file with the .pil file-extension. This file is referred to as the PIL specification. PIL compiler The polynomial identities need to be compiled into a language that prover can understand. A special PIL compiler called pilcom is translate the PIL specification into .json file called Constraints file. !Figure: Pilcom The constraints .json file contains a list of all the constraints and additional information about the polyomials, and it can be consumed by the prover. The repository pilcom-vscode contains a PIL syntax highlighter for VSCode. Proof generation and verification The transformation of constraints into polynomial indentities, captured in the PIL specification file, and the compilation of the PIL specification file into the constraints .json file happens in the zkEVM's cryptographic backend. The cryptographic backend receives the outputs of the executor: that is, both the witness and fixed columns, as well as the constraints (uniquely describing the execution matrix that models the input program). The prover, which is part of this cryptographic backend, is in charge of generating proofs. The verifier, which is the zkEVM's component responsible for verifying proofs, takes as inputs a proof and its corresponding publics. The publics, which is short for public values, is a set of values known by both the prover and the verifier. The verifier uses a specialized verification algorithm to check validity of the proof, for the provided publics. And its output is either an OK for an accepted proof, or a KO for a rejected proof. The figure below, is a simplified scheme that outlines the systematic process of generating and verifying proofs. !Figure: Cryptographic backend and Verifier The current zkEVM setup relies on a backend cryptographic system where the verifier is an algorithm called FFlonk. Further details are covered later in this document. Since proofs are compact and their verification demands minimal resources, a smart contract can carry out verification of proofs. In reality, a smart contract in L1 implements the verifier algorithm. The gas cost for this verification is approximately 200,000 gas units. Public and private inputs Zero-Knowledge technology enables separation between public and private values among the cells of the execution trace. This provides the capability to designate certain variables for holding private data while others carry public information. Example: (Proving knowledge of a pre-image of a digest) Consider, for instance, a password protocol of a simplified public-key infrastructure (PKI). A user keeps their password as a secret, while they are authenticated by the hash-value of their password (the hash-value acts like a 'public-key'). Proof that one knows the password corresponding to a given hash-value is being able to supply the password that reproduces the hash-value. Typically, this is done by internally applying a designated hash function on the supplied password. $$ \mathtt{hash-value} := \textbf{Hash}(\mathtt{password}) $$ The security of the protocol relies on such a hash function being collision-resistant. That is, the probability of producing the same hash-value from a different private-key is extremely small. This example illustrates the need to distinguish between publics (i.e., the hash-value) and the private inputs (i.e., the password) in order to achieve some privacy. Verfier's inputs Publicly known values, or publics, are not only important in the proving phase, but also in the verification phase. As seen in the previous figure above, publics form part of the verfier's inputs. And, similar to the above pasword protocol, these publics are uniquely related to the executor's input program. By default, all values in PIL are considered private. However, any specific value can be made public by using the keyword public. !Figure: The figure above illustrates a hash pre-image proof, where $\texttt{hashvalue}$ is public, while $\texttt{hashpre-image}$ remains confidential. The initial cell of the execution trace (colored red) signifies a secret input, which is put through a sequence of instructions (designed to model the hash function $\textbf{Hash}$), and culminates in the production of the hash output. Subsequently, the cryptographic backend generates a proof π and public values, and these inputs to the verifier. Next PIL version: PIL2 Currently, we are in the process of developing a new version of PIL, referred to as PIL2. PIL2 is designed to operate with a more potent cryptographic backend capable of generating an adequate number of subexecution traces to accommodate the whole batch processing without running out of rows. Additionally, we are collaborating with other projects within the zk projects at Polygon to establish a standardized format for the PIL output file, named pilout. # processing-l2-blocks.md: In this document we discuss the differences between the Dragonfruit upgrade, which comes with the executor fork-ID 5, and the Etrog upgrade associated with fork-ID 6. The key differences between the two Polygon zkEVM upgrades are mainly related to the definition of the L2 block and timestamps. In the Dragonfruit upgrade, - An L2 block is defined to contain only one transaction, resulting in as many blocks per batch as there are transactions. - Timestamps are not assigned to blocks but to batches, which means each batch typically contains more than one block. Since the timestamp is part of batch data instead of block data, it is shared among all the blocks within the batch. Although the Dragonfruit approach minimizes delay, it has the following drawbacks: - It leads to a bloated database due to the large number of L2 blocks created. - It causes breaks in dApps that are configured with block-per-timestamp settings, as they rely on timestamps for proper timing of smart contract actions. The Etrog upgrade addresses these two issues by allowing multiple transactions per block and assigning a unique timestamp to each block rather than to each batch. It also introduces a timeout of a few seconds or milliseconds, during which the sequencer waits for transactions while creating a block. To change the timestamp from one block to the next, the sequencer uses a special transaction as a new block marker, called changeL2Block. The figure below displays the Etrog block structure within a batch. !Figure: Several blocks in a batch In this document, we delve into the processing of a block in both Dragonfruit and Etrog upgrades. The 0x5ca1ab1e smart contract Starting from the proving system's point of view, recall that, - The aggregator receives several inputs, such as the $\texttt{oldStateRoot}$, $\texttt{initNumBatch}$ and $\texttt{oldAccInputHash}$. - The accumulated input hash $\texttt{accInputHash}$ is a recursively computed cryptographic representative of several batch data, including the hash of all the L2 transactions within previous batches and the last batchâ€™s sequencing timestamp. Also, due to its recursive nature, the $\texttt{accInputHash}$ incorporates the previous accumulated input hash $\texttt{oldAccInputHash}$ as part of its hashed data. The figure below depicts the Aggregator schema in the Dragonfruit context. !Figure: Aggregator schema - Dragonfruit Pertaining to the figure above, one may ask: How does the proverâ€™s processing system access and secure the block number (which, in the Dragonfruit context, is equivalent to the transaction number). The answer is: This information is included within the L2 state. Specifically, the data is held in the storage slot 0 of an L2 system smart contract, which is deployed at the address 0x5ca1ab1e. After processing a transaction, the ROM writes the current block number into this specific storage location. As depicted in the figure below, the L2 system smart contract deployed at address 0x5ca1ab1e stores the number of the last processed block at slot 0. Henceforth, during each batch processing, the system records all block numbers it contains. !Figure: The L2 5ca1ab1e smart contract Since the smart contract deployed at the address 0x5ca1ab1e is frequently referred to throughout this document, we call it by its address. i.e., We refer to it as the 0x5ca1ab1e system smart contract. The BLOCKHASH opcode In the EVM, the BLOCKHASH Opcode provides the keccak-256 digest of the Ethereum L1 block header, which includes: root of the state trie, root of transactions trie, root of receipt trie, logs, gas used, gas limit, block number, timestamp, etc. A complete list of all parameters stored in an Ethereum block and block header is given in the Ethereum organisation documentation. You can use the Geth library to compute an Ethereum block hash. See the figure below for an example of an Ethereum L1 block header reflecting some of these parameters. !Figure: Ethereum L1 Block Header The Polygon zkEVM RPC provides methods for obtaining the L2 $\texttt{BLOCKHASH}$ in the Ethereum-style. You can find a list of specific zkEVM endpoints in the zkEVM node repository. One such method is zkevmgetFullBlockByHash which, given its Ethereum-like block hash, retrieves a block with additional information. Dragonfruit (ForkID 5) Following Ethereumâ€™s philosophy, there is a need to keep track of every state change between blocks. In the Dragonfruit setting, this is equivalent to tracking state changes per transaction. For the sake of security, all the state changes must be accounted for. Polygon zkEVM therefore stores the state root after processing each transaction. In each case the state root is stored in a designated slot: The slot 1 of the 0x5ca1ab1e smart contract, as illustrated in the figure below. With this approach, the state root is stored for each transaction within a batch, allowing for precise monitoring of the entire batch processing at the transaction level. L2 BLOCKHASH In the ForkID 5 context, the L2 BLOCKHASH opcode provides only the state root when executed by smart contracts. We define this particular output of the L2 BLOCKHASH opcode as the native block hash, and provide the state root accessing the 0x5ca1ab1e smart contract. L2 system smart contract 0x5ca1ab1e stores the last state root $SR{k-1}$, after processing a block, at slot 1. !Figure: Storing at slot 1 of 5ca1ab1e Up until, and including, the Dragonfruit upgrade, the Polygon zkEVM processing did not store any extra parameters about block execution inside 0x5ca1ab1e. Processing L2 blocks The question is: When to write the state root in the 0x5ca1ab1e system smart contract? Consider the figure below for a schematic diagram of how the new state root is updated and written in the 0x5ca1ab1e system smart contract when processing L2 blocks. Firstly, denote the last block among the completely processed blocks of a batch by $kâ^'1$. And hence denote the state root at this point by $SR{kâ^'1}$. After completing the processing of the last block $k-1$, the zkEVM updates the slot 1 of the 0x5ca1ab1e smart contract with the current state root. But since updating a storage slot of an L2 smart contract actually changes the L2 state, it leads to a new state denoted by $SR'{kâ^'1}$. At this point, we start processing the new batch. !Figure: New state update schema It is important to note that the state root $SR{kâ^'1}$ (i.e., the current state before both the storing of the state root $SR{kâ^'1}$ in the 0x5ca1ab1e contract and starting to process any new block) and the state root $SR'{kâ^'1}$ do not match. Yet, when the new block k is executed, the state root becomes SRk. And, upon writing the state root SRk to the 0x5ca1ab1e contract, the state root becomes $SR'k$. This pattern continues with subsequent blocks. Therefore, it can be misleading to rely on the state root stored at slot 1 of the 0x5ca1ab1e contract as the representative of the L2 state (i.e., The root of the state at the end of the execution of the previous block) because the actual corresponding state root is the state root after updating the contract. Consequently, in the Dragonfruit upgrade, the value obtained when BLOCKHASH is called is not precisely the same state root obtained just before the execution of the new block begins. This mismatch issue is addressed in the Etrog upgrade. L2 native vs. L2 RPC Ethereum-like BLOCKHASH In Ethereum, the block header is secure because it is computed and validated by all the nodes within the network. However, in the Polygon zkEVM, the prover is the only entity responsible for proving that the parameters related to block execution are correct, and these parameters form part of the state. Ethereum takes the approach that block parameters, providing information about execution of transactions in each block, are hashed to obtain the block hash. And, the resulting state root is one of these parameters. Since the aim is to prove that the block hash computation and its parameters are correct, the native block hash in the Polygon zkEVM context has to be the L2 state root. The zkEVM prover is in charge of proving that the changes in the L2 state root are correctly performed. So, if we want to provide a verifiable proof of the execution parameters of a block (such as gasUsed, transaction logs, etc.,) we have to work these parameters into the Polygon zkEVM processing, including them in the L2 state. Incorporating block execution parameters into the L2 state is facilitated through the 0x5ca1ab1e smart contract. Thus, the L2 state root is a hash that contains all the parameters that provide information about block execution. The figure below depicts the differences. !Figure: Blockhash - Ethereum vs. zkEVM The L2 native block hash is computed differently from the Ethereum block hash. Here are some of the differences: - As part of the block execution, zkEVM-specific parameters like each transaction's $\texttt{effectivePercentage}$ is included. - Poseidon hash function is used instead of Keccak-256. - Transactions data is hashed with linear Poseidon. In the RPC, the method zkevmgetNativeBlockHashesInRange returns the list of â€œnative block hashesâ€. That is, the list of L2 state roots. Etrog upgrade (Fork-ID 6) In the zkEVM Etrog, similar to the Ethereum setting but not identical, additional data related to the L2 block processing is secured via the 0x5ca1ab1e smart contract. In particular, the L2 system smart contract 0x5ca1ab1e stores, in: - Slot 0: The block number of the last processed block, as in the Dragonfruit upgrade. - Slots hash(1|blockNum): These slots are encoded as Solidity mappings, to store all state roots indexed per block number. - Slot 2: The timestamp of the last processed block, because now we have a timestamp for each block. - Slot 3: The root of a Read-Write Merkle tree called $\mathtt{BlockInfoTree}$, which contains information about the execution of the last processed block. The figure below depicts a schematic diagram of the 0x5ca1ab1e contract storage in the Etrog upgrade. !Figure: 5ca1ab1e slots - Fork Etrog BlockInfoTree Next we describe the $\mathtt{BlockInfoTree}$, together with its contents, keys, and values. The $\mathtt{BlockInfoTree}$ is a Read-Write Merkle tree containing information about the execution of the last processed block. Observe that the $\mathtt{BlockInfoTree}$ is unique for each block. Contents The $\mathtt{BlockInfoTree}$ stores the specific data associated with each transaction, simply referred as the transaction data, and denoted by $\texttt{txData}$. So, $\texttt{txData}$ is an array of data: $$ \texttt{txData} = \texttt{(nonce,gasPrice, gasLimit, to, value, data, from)} $$ But each L2 transaction's data is stored as its cryptographic representation as follows: $$ \texttt{transactionHashL2 = LinearPoseidon(txData)} $$ The $\texttt{from}$ field is actually computed from the transactionâ€™s signature using ecRecover. Each executed transaction includes a byte parameter called $\texttt{effectivePercentage}$, which is specific for Polygon zkEVM. This parameter is also stored in the $\mathtt{BlockInfoTree}$. The $\mathtt{BlockInfoTree}$ also stores the following data, coming from L2 transaction execution: - $\texttt{status}$ - $\texttt{receiptData}$ - $\texttt{cumulativeGasUsed}$ - $\mathtt{linearPoseidon(log\0\data)}$ - $\mathtt{linearPoseidon(log\1\data)}$ - $\texttt{...}$ - $\mathtt{linearPoseidon(log\N-1\data)}$ Among the data stored in the $\mathtt{BlockInfoTree}$, there's information common to all blocks, consisting of the following data: 1. Data stored in or updated from the L2 state, obtained from the 0x5ca1ab1e smart contract: â€“ $\texttt{Previous blockhashL2}$ (i.e., The previous L2 state root). â€“ $\texttt{blockNumber}$. 2. Data from each L2 block, obtained from L1 as the proof input: â€“ $\texttt{timestamp}$ is from the changeL2block transaction. â€“ $\texttt{globalExitRoot}$ for the bridge. â€“ $\texttt{blockHashL1}$, which is the L1 block hash when the globalExitRoot parameter was recorded by the L1 contract PolygonZkEVMGlobalExitRoot.sol. In Solidity, this is done using: $\texttt{blockhash(block.number - 1)}$. 3. Data from the data of sequenced batches obtained from L1 as the proof input: â€“ $\texttt{coinbase L2}$. 4. Data computed from the block execution: â€“ $\texttt{gasUsed}$. 5. Other parameters: â€“ $\texttt{gasLimit}$. Although infinite, a single transaction is limited to a maximum of 30M gas in the zkEVM. Below is the schematic diagram of the Aggregator in the Etrog upgrade. !Figure: Aggregator schema - Fork Etrog Keys and values How to store all previous block-related and transaction-related data in the $\mathtt{BlockInfoTree}$? The $\mathtt{BlockInfoTree}$, being unique to each block, operates as a Read-Write sparse Merkle tree and functions similar to the L2StateTree key-value structure. Itâ€™s also constructed using the Poseidon hash function. The keys utilized to position each data piece within the tree are also derived by applying the Poseidon hash function. The output of the Poseidon hash function is given by: $$ \begin{aligned} &\mathtt{\big(out[0], out[1], out[2], out[3]\big)\ } \\ &\mathtt{\ = Poseidon\big(c[0], c[1], c[2], c[3]; in[0], in[1], in[2], in[3], Â· Â· Â· , in[7]\big)} \\ \end{aligned} $$ where for $i \in \{0, 1, 2, 3\}$, each $\mathtt{c[i]}$ is a capacity element, and $\mathtt{in[i]}$ is an input element. Also, each Poseidon parameter is a field element in the Goldilocks field $\mathbb{F}p$ where $p = 2^{64} â^' 2^{32} + 1$. Block-related data When computing keys of block-related data, the Poseidon function is used as follows: $$ \mathtt{ outputs = Poseidon(0; INDEX\BLOCK\HEADER, 0, 0, 0, 0, 0, SMT\KEY, 0) } $$ where $\mathtt{SMT\KEY}$ is fixed to the value 7 (i.e., $\mathtt{SMT\KEY} := 7$) and $\mathtt{INDEX\BLOCK\HEADER}$ is used to distinguish different block-related data as listed below: - $\mathtt{INDEX\BLOCK\HEADER} = 0$, for the previous block hash. - $\mathtt{INDEX\BLOCK\HEADER} = 1$, for the coinbase address. - $\mathtt{INDEX\BLOCK\HEADER} = 2$, for the block number. - $\mathtt{INDEX\BLOCK\HEADER} = 3$, for the gas limit. - $\mathtt{INDEX\BLOCK\HEADER} = 4$, for the block timestamp. - $\mathtt{INDEX\BLOCK\HEADER} = 5$, for the Global Exit Root. - $\mathtt{INDEX\BLOCK\HEADER} = 6$, for the L1 block hash. - $\mathtt{INDEX\BLOCK\HEADER} = 7$, for the gas used. Transaction-related data When computing keys of transaction-related data, the Poseidon function is used as follows: $$ \mathtt{leafKey = Poseidon(0; txIndex[0], txIndex[1], Â· Â· Â· , txIndex[4], 0, SMTKEY, 0)} $$ where the transaction index is hashed within the block, and $\mathtt{SMT\KEY}$ takes any of the following values: - $\mathtt{SMT\KEY} = 8$, for the transaction data hash. - $\mathtt{SMT\KEY} = 9$, for the transaction status. - $\mathtt{SMT\KEY} = 10$, for the transaction cumulative gas used. - $\mathtt{SMT\KEY} = 11$, for the transaction logs. - $\mathtt{SMT\KEY} = 12$, for the transaction's effective percentage. Finally, given multiple logs per transaction, the key for logs is computed as follows: $$ \mathtt{leafKey = Poseidon(logIndex; txIndex[0], txIndex[1], Â· Â· Â· , txIndex[4], 0, SMTKEY, 0)} $$ where the index of the log is hashed within the block. Processing L2 blocks How are L2 blocks processed in the Etrog upgrade? The diagram below is useful in explaining the process. !Figure: Processing L2 - Fork Etrog At the end of block execution, the zkEVM writes the last block number, the timestamp and the block info root, but does not write the state root. So, as shown in the figure above, at the end of executing block $kâ^'1$, the 0x5ca1ab1e smart contract stores four parameters: $$ \begin{aligned} &\mathtt{LAST\ BLOCK} = kâ^'1 \\ &\mathtt{SR = SR{kâ^'2}} \\ &\mathtt{TIMESTAMP{kâ^'1}} \\ &\mathtt{BLOCK\ INFO\ ROOT{kâ^'1}} \\ \end{aligned} $$ with the state root $\mathtt{SR}$ still reflecting the root corresponding to block $k-2$. Next, the system starts processing a new block, which starts with the changeL2Block transaction, as the first transaction to be processed in a block. The proving system provides the $\mathtt{oldStateRoot = SR{kâ^'1}}$, and the initial step of processing a block in the ROM is to record it in the 0x5ca1ab1e smart contract. Subsequently, and in contrast to what happened in Dragonfruit upgrade, every transaction containing the BLOCKHASH opcode now provides the correct state root, $\mathtt{SR{kâ^'1}}$. As shown in the figure below, while transactions within the block are processed, the zkEVM not only updates the L2 state but also adds the information to the $\mathtt{BlockInfoTree}$. Upon completing the execution of the block, the root of this tree is written into 0x5ca1ab1e. When executing all transactions within the block, the 0x5ca1ab1e contract is updated again. And the whole process is repeated. !Figure: BlockInfoTree - processing one transaction The $\mathtt{BlockInfoTree}$ is built while processing each transaction within the block. It is important to mention that the state roots of the block info trees are linked, as depicted in the figure below. They are linked by storing the state root of the $\mathtt{BlockInfoTree}{kâ^'1}$ in the next one, $\mathtt{BlockInfoTree}k$. This way, the whole history is always captured in the current $\mathtt{BlockInfoTree}$. And it therefore enables proving any previous data by recursively using Merkle proofs. !Figure: BlockInfoTree root update # prover-and-verifier-recap.md: The Prover, shown in the figure below, is the component within the Proving System which is in charge of generating a proof for the correct execution of some program. The program, which takes care of the computational aspect of the computation, is written in a language called zkASM, developed by the Polygon team. Providing also the private and the public inputs, the Executor component within the Prover is able to generate the execution traces designed to model the willing computation. The executor produces two binary files: one containing the fixed columns, which should only be generated once if we do not change the computation itself, and the other containing the witness columns, which vary with inputs and thus need to be generated anew for each proof. The files containing the pre- processed fixed columns and the processed witness columns for the zkEVM are temporary stored in binary files and are quite large, consisting on more than 100Gb. Subsequently, the cryptographic backend of the prover, in conjunction with the compiled PIL constraints through pilcom and the execution trace binary files generated earlier, can produce the proof and provide the public values, both inputs and outputs, for the verifier. !Figure: The figure above depicts the schema of the prover component. The Prover component has 3 subcomponents: - The Executor, which is in charge of populating the execution trace based on some computation with values depending on some inputs. - The PIL compiler or pilcom, which compiles PIL files into JSON files prepared to be consumed by the Prover. - The cryptographic backend, which takes the output of both the Executor and the pilcom and generates the proof and the publics for the verifier. Posteriorly, as depicted below, the verifier uses both the proof and the public values to check if the prover has performed the computation in a correct way. !Figure: The verifier utilizes both the proof and the public values to validate the correctness of the computation performed by the prover. # proving-inputs.md: Generating a proof for the valid processing of a single batch requires multiple inputs. In addition to the data from each transaction in the processed batch, other inputs are necessary to ensure the network's overall security. For example, some inputs need to be sent to the smart contract. Central to generation of proofs and their verification are the verifier smart contract and the prover, as well as their interaction. This document, therefore, explores what the prover transmits to the verifier smart contract and why. In the zkEVM context, the focus is on generating succinct[^1] proofs rather than privacy concerns. Consequently, L2 transactions and L2 state data are public. We therefore delve into the initial design of the proving system, which operates without private inputs. Public inputs Recall that the prover generates a proof, and sends the proof and an array of public inputs, denoted by $\texttt{[publics]}$ to the smart contract. The figure below depicts this. !Figure: The public inputs consist of the following data: - batchData which is the data of all the L2 transactions in the batch being proved. - currentStateRoot referring to the current L2 state root. - proverAccount which is the prover's account due to receive rewards. This account is attached to the proof so as to avoid successful plagiarism of the proof. - timestamp specifies the time at which the proof was generated. - forkId is the current version of the L2 EVM being used. - chainId is the identifier of the chain for which the proof is being generated. The protocol is designed with the capability to host multiple layer 2 networks. The public output of this process is a new L2 state root, denoted by newStateRoot. The prover does not send all these public inputs to the L1 smart contract. Some inputs are already stored in the contract. These include: currentStateRoot, forkId, and chainId. The rest of the inputs need to be included in the calldata of the transaction sent to the L1â€™s smart contract. That is, batchData, timestamp, newStateRoot. Since the proverAccount is included in the L1 transactionâ€™s signature, it need not be explicitly provided. Finally, the prover also sends the proof of correct execution of all L2 transactions within the batch. [^1]: The term succinct refers to the size of the proof being very small compared to the size of the statement or the witness (i.e., the size of the computation itself). # secondary-sms-lookup-tables.md: Let's assume that the cryptographic backend only allows defining constraints with additions and multiplications. Consider implementing an exponentiation operation, denoted by EXP. This can be implemented by performing a few multiplications. An execution trace can be set to only 3 columns A, B and C, each defined by the following constraints: 1. Column A represents the base, and it is defined by $a' = a$. 2. Column B stores the decreasing exponent, so $b' = b âˆ' 1$. 3. Column C stores the intermediate results. That is, $c' = c \cdot a$, For instance, a portion of the execution trace (that implements the operation $2^5 = 32$â€‹) is shown in table below: !Figure: The above execution trace models the exponentiation operation 2^5. The blue cells represent the input values, and the green cell contains the output of the computation. When implementing this operation in the main execution trace, itâ€™s important to know that each EXP operation will require multiple rows. In fact, the exact number of rows required for EXP depends on the exponent. This approach easily leads to a complicated set of constraints, together with a huge number of used up by the operation. One way to solve to this problem is to utilize secondary execution traces tailor-made for specific computations like the EXP. Such an approach assumes existence of the so-called main execution trace, to which a few specialist secondary execution traces are linked. The cryptographic backend utilizes a mechanism called lookup argument to link these execution traces. The lookup argument, in particular, provides the constraints required to check whether certain cells of a row match some cells in a row of another execution matrix. Linking execution traces via lookup arguments Let us now explore how the above-mentioned approach, regarding having the main execution trace which is linked to secondary execution traces, actually works. We continue with the EXP example, where only two execution traces are considered: - The main execution trace and - A secondary execution trace especially created for the EXP operation. The main execution trace In the main execution trace, shown in the table below, each EXP operation will occupy just one row. For the first EXP operation, the input values 2 and 5 yields the result 32. $$ \begin{aligned} \begin{array}{|c|c|c|c|c|c|}\hline \bf{\texttt{ A }} & \bf{\texttt{ B }} & \bf{\texttt{ C }} & \bf{\texttt{EXP}} \\ \hline \cdots & \cdots & \cdots & 0 \\ \hline 2 & 5 & 32 & 1 \\ \hline \cdots & \cdots & \cdots & 0 \\ \hline \cdots & \cdots & \cdots & 0 \\ \hline \cdots & \cdots & \cdots & 0 \\ \hline \cdots & \cdots & \cdots & 0 \\ \hline 3 & 2 & 9 & 1 \\ \hline \cdots & \cdots & \cdots & 0 \\ \hline \cdots & \cdots & \cdots & 0 \\ \hline \text{} & \text{} & \text{} & \text{} \\ \hline \end{array} \end{aligned} $$ In the main execution trace depicted above, we observe that the value 1 in the EXP column flags when an EXP operation is performed. A value 1 in the EXP column indicates when (i.e., in which row,) to check whether the value of C can be derived by exponentiating the value in column A to the power of the value in column B. The correctness of the result will be validated in a secondary execution matrix. Remark An execution trace can be viewed as a set of states (in some state machine), where each row represents a state. For this reason, the terms execution trace and state machine are used interchangeably. A secondary state machine The secondary state machine operates in the same manner described for the 2^5 EXP example. In this case, the columns are described as follows: 1. Column A represents the base. Observe that this column is allowed to contain several values as we may check several EXP operations in the same state machine for as long as we do not consume more rows than the fixed maximum. 2. Column B stores the exponent. 3. Column C stores the intermediate results. 4. Column D stores a decreasing counter. In each EXP operation, it starts with the same value as the exponent and keeps decreasing until arriving 1, meaning that the current operation has been finished. 5. The EXP column flags when the operation is actually finished. Rows with EXP equal to 1 are the ones where we can state consistency of EXP operations of the main state sachine via a lookup argument. The connection between the two state machines is established through a lookup argument. Whenever EXP's selector value in the main state machine is 1â€‹, a search is made (looked up) in the secondary state machine for a row with the same values in the columns A, B and C. Only rows with the value 1 in the EXP selector column are searched. The presence of a row with the exact same values in the secondary state machine as in the main state machine proofs that the EXP operation was correctly performed. Otherwise, the prover would have cheated. !Figure: The linkage between the main state machine and the EXP secondary state machine is done by checking equality between yellow rows of both state machines. The zkEVM's Main state machine In the Main SM, we just put the inputs/outputs, that we call free, in a single row. Recall that there is a PIL compiler that reads a PIL specification file and compiles it to an output file with a list of constraints and a format that can be consumed by the prover to generate a proof. In the PIL language, the state machines, that is, the different sub-execution matrices linked together via lookup arguments, are called namespaces. In the zkevm-proverjs repository, you can find the PIL specification of the whole zkEVM. There are several files, one for each state machine, such that binary.pil, that is responsible for constraining binary operations (such that additions of 256 numbers represented in base 2 and equality/inequality comparators), or mem.pil, which is responsible for managing memory-related opcodes. The Main State Machine is constrained in the main.pil, which serves as the main entrypoint for the whole PIL of the zkEVM. Final remarks and future improvements The columns of each state machine are defined by the design of its corresponding execution trace. Due to constraints in our existing cryptographic backend, it is mandatory that all state machines share the same number of rows. The computation of an L2 batch can have branches and loops and hence, each L2 batch execution can use a different number of operations in the zkEVM. Consequently, the number of rows utilized by each state machine depends on the specific operations carried out during batch execution. Since the number of rows is fixed (and the same for all state machines) we can have unused rows. But, what is more important is that obviously, the size of the computation being proved must fit in the execution trace matrices available !Figure: Due to the specific cryptographic backend used in the prover, all the same state machines should have the same fixed amount of rows. # the-executor-comp.md: The executor is responsible for generating a correct execution trace from a given set of inputs. As shown in the figure below, the executor takes some inputs and generates an execution trace. !Figure: There are two approaches in which the executor component can be implemented, depending on whether we prefer: - A single-computation executor that only runs a specific program, or - A general purpose executor which runs several computations. Single-computation executor A single-computation executor is tailor-made for a particular computation. This means any change in the computation will require the executor to be modified. Although this type of executors are computationally faster, they are not easy to change, test, or audit. A single-computation executor is also called a native executor. An electronic analogy to a native executor is an Application Specific Integrated Circuit (ASIC). An ASIC is a circuit specifically designed to run a single computation efficiently. Since a native executor always performs the same operation with different sets of inputs, the fixed column of the execution trace can be "hardcoded" or pre-processed. A native executor for computing $[(x0 +x1)Â·4]Â·x2$, as in our previous example, can be depicted as in the figure below. !Figure: General-computation executor A general-computation executor can run a wide variety of programs. In this approach, the executor component not only reads inputs, as in the previous approach, but also interprets each program. A program therefore, is like a set of instructions that guides the executor on how to compute the execution trace for a specific computation. In the general-computation executor case, we therefore add programs written in the zero-knowledge Assembly (zkASM[^1]) language, as inputs to the Executor. !Figure: A general-computation executor takes as inputs: - Input values: For instance, $x := (x0, x1, x2)$ - An ordered set of instructions: For instance, $\big(\texttt{ADD},\ \texttt{TIMES4},\ \texttt{MUL} \big)$. This means the executor corresponding to the above example, where the computation $[(x0 + x1) Â· 4] Â· x2$ was executed, is in fact a general-computation executor. Example (Same executor, different program) Note that the same executor, used in the above example, can be used to execute any other program that involves the available instructions: $\texttt{ADD},\ \texttt{TIMES4},\ \texttt{MUL}$â€‹. Consider the following computation: $$ (x0 \cdot 16)Â·x1 $$ which translates to the following program, given as an ordered set of instructions: $$ \big(\texttt{TIMES4},\ \texttt{TIMES4},\ \texttt{MUL} \big) $$ Since this program involves available instructions, the same executor of the previous example can be used to execute this program. The corresponding execution trace, for when the input value $x := (2,3,\)$, is given below: !Figure: Comparing the two executor types Each executor type has its own pros and cons. Single-computation executors do not need to read instructions from programs as they always perform the same operations. They are fast as they do not have to read or compile assembly codes. This type of executors are therefore efficient in generating execution traces. The main drawback of single-computation executors is their inflexibility, making them difficult to modify, test, or audit. In contrast, general-computation executors are much slower than their counterparts as they need to read and compile input programs written in zkASM. The biggest advantage with general-computation executors is flexibility. It is more efficient to keep modifying the assembly code than the whole executor. The pros and cons of the two executor types is summarised in the table below. | Executor type | Pros | Cons | | :-----------------: | :-----------: | :-----------: | | Single-computation | Faster | Less flexible | | General-computation | More flexible | Slower | Programs written in zkASM As mentioned earlier, the executor takes as inputs programs that are written in the zero-knowledge Assembly (zkASM) language. zkASM is a language developed by the Polygon Hermez team, and it is used to write instructions for the general-computation executor. You can find a syntax highlighter, for VS Code, in the $\texttt{zkasmcom-vscode}$ repository. Below is a zkASM code snippet, which is taken from the zkEVM project. !Figure: zkASM Language zkASM comes with a compiler called $\texttt{zkasmcom}$. This $\texttt{zkasmcom}$ compiler, - First reads a program written in zkASM, which is a .zkasm specification file. - Then compiles and outputs an instruction file, which is a .json file containing a list of steps and instructions the executor must follow when creating the execution trace. !Figure: zkASM Compiler The general-computation executor approach suits the zkEVM context best, because both the EVM and the zkEVM evolve. Also, it is more efficient to keep modifying the assembly code than the whole executor. The zkASM program that processes EVM transactions is called zkEVM ROM (as in "Read Only Memory") or simply the ROM. By changing the ROM, we can make the L2 zkEVM more and more closer to the L1 EVM. Hence, we have different versions of the zkEVM ROM. Each of these versions will be denoted with a unique identifier called forkID. It is worth mentioning that another advantage of using a ROM-based approach is that we can test small parts of the assembly program in isolation. Since each approach to executor design serves a different purpose, the zkEVM utilizes both approaches. However, the single-computation executor is still a work in progress. [^1]: "zkASM" is read as 'zk-assembly' # bridging.md: The unified bridge transfers assets and messages between networks (L1, L2) by calling bridge and claim functions on the PolygonZkEVMBridgeV2.sol contract whose deployment address is an upgradeable proxy. Interaction with the actual contract happens via the proxy. This contract is deployed on L1 and there is also one deployed on every L2 network. It communicates closely with an exit root manager contract specific to L1 or L2. !Polygon bridge contracts !!! tip - Notice that the L2 bridge content has a function for updating the global exit root: setGlobalExitRoot(...). Bridge and claim The main functions of the bridge are: - bridgeAsset(...) - bridgeMessage(...) - claimAsset(...) - claimMessage(...) L1 to L2 To bridge assets from L1 to L2, the sender first transfers the token into the bridge by locking the asset on the origin network (L1). When executing claimAsset, the bridge smart contract mints a wrapped token of the original asset on the destination network (L2). The wrapped token is a generic ERC20. Once minted, the recipient can claim the token on the destination network (L2). The data below is transaction data (represented by a leaf node in an exit tree) and comes from an example L1 to L2 transaction recorded on the L2 zkEVM chain after a successful bridgeAsset call. | Index | Parameter name | Data type | Example value | |------|---------------------------|---------|--| | 0 | destinationNetwork | uint32 | 1 (zkEVM network) | | 1 | destinationAddress | address | 0x2944119d60465AE2f4E6C587fCc669214a5094E1 | | 2 | amount | uint256 | 20270613020262720 | | 3 | token | address | 0x0000000000000000000000000000000000000000 (Ether) | | 4 | forceUpdateGlobalExitRoot | bool | true (indicates if new global exit root is updated) | | 5 | permitData | bytes | (Raw data of the call permit of the token) | L2 to L1 To send an asset from L2 to L1, the wrapped token is first burnt on the L2 network. When executing claimAsset, the bridge smart contract unlocks the original asset on the origin network (L1) ready for claiming. The data below is transaction data (represented by a leaf node in an exit tree) and comes from an example L2 to L1 claim transaction which also posts the local and network exit roots and root proofs used for verification on L1. | | Parameter name | Data type | Example value | |----|-----------------------|------------|---| | 0 | smtProofLocalExitRoot | bytes32[32] | 0xdaaaac32944200f40bbf1e208472... | | 1 | smtProofRollupExitRoot | bytes32[32] | 0xf73a9a58cf58b6ba6a4cc7b4951a... | | 2 | globalIndex | uint256 | 48352 (used for synchronizing) | | 3 | mainnetExitRoot | bytes32 | 0x22e6d13ed71c26a403b8bae97755fc215744bfa490d108aa8d14386fef41de02 | | 4 | rollupExitRoot | bytes32 | 0x118318620b268cc322953926c3b45092e573af034ddf75c143456b2886a844ef | | 5 | originNetwork | uint32 | 0 | | 6 | originTokenAddress | address | 0x0000000000000000000000000000000000000000 | | 7 | destinationNetwork | uint32 | 0 | | 8 | destinationAddress | address | 0x5251b3304d1bA5834fd227c2842AA82aC50412E6 | | 9 | amount | uint256 | 67000000000000000 | | 10 | metadata | bytes | (abi encoded metadata if any, empty otherwise) | Transaction flows in depth L1 to L2 1. If a call to bridgeAsset or bridgeMessage on L1 passes validation, the bridge contract appends an exit leaf to the L1 exit tree and computes the new L1 exit tree root. 2. The global exit root manager appends the new L1 exit tree root to the global exit tree and computes the global exit root. 3. The sequencer fetches the latest global exit root from the global exit root manager. 4. At the start of the transaction batch, the sequencer stores the global exit root in special storage slots of the L2 global exit root manager smart contract, allowing L2 users to access it. 5. A call to claimAsset or claimMessage provides a Merkle proof that validates the correct exit leaf in the global exit root. 6. The bridge contract validates the caller's Merkle proof against the global exit root. If the proof is valid, the bridging process succeeds; otherwise, the transaction fails. L2 to L1 1. If a call to bridgeAsset or bridgeMessage on L2 passes validation, the bridge contract appends an exit leaf to the L2 exit tree and computes the new L2 exit tree root. 2. The L2 global exit root manager appends the new L2 exit tree root to the global exit tree and computes the global exit root. At that point, the caller's bridge transaction is included in one of batches selected and sequenced by the sequencer. 3. The aggregator generates a zk-proof attesting to the computational integrity in the execution of sequenced batches which include the transaction. 4. For verification purposes, the aggregator sends the zk-proof together with all relevant batch information that led to the new L2 exit tree root (computed in step 2), to the consensus contract. 5. The consensus contract utilizes the verifyBatches function to verify validity of the received zk-proof. If valid, the contract sends the new L2 exit tree root to the global exit root manager in order to update the global exit tree. 6. claimMessage or claimAsset is then called on the bridge contract with Merkle proofs for correct validation of exit leaves. 7. The bridge contract retrieves the global exit root from the L1 global exit root manager and verifies validity of the Merkle proof. If the Merkle proof is valid, the bridge completes. Otherwise, the transaction is reverted. L2 to L2 1. When a batch of transactions is processed, the bridge contract appends the L2 exit tree with a new leaf containing the batch information. This updates the L2 exit tree root. 2. The bridge contract transmits the L2 exit tree root to the L2 global exit root manager. The L2 global exit root manager, however, does not update the global exit tree at this stage. 3. For proving and verification, the zk-proof-generating circuit obtains the L2 exit tree root from the L2 global exit root manager. 4. Only after the batch has been successfully proved and verified does the L2 global exit root manager append the L2 exit tree root to the global exit tree. As a result, the global exit root is updated. 5. The zk-proof-generating circuit also writes the L2 exit tree root to the mainnet. The L1 bridge contract can then finalize the transfer by using the claim function. # index.md: The LxLy bridge is an interoperability solution aimed at enabling cross-chain communication among networks. It facilitates interaction between two L2 chains or between an L2 chain and Ethereum as the L1. The LxLy bridge SC (or PolygonZkEVMBridgeV2) is an improved and a more robust version of the zkEVM bridge deployed in the Polygon zkEVM mainnetbeta version. Its modular design capacitates projects to deploy their own rollups and connect them to the Polygon ecosystem. Ideal attributes The LxLy bridge is deployed in the Polygon ecosystem to enable the following features and functionalities: - Accessibility: Projects can request creation of a rollup and connect it to the Polygon ecosystem. - Unified liquidity: All rollups can connect through the same bridge, enabling L2 to L2 bridges. - Polygon CDK: All the rollups can use the same stack. - Rollup-project oriented: Every project can have its own rollup. - Cross-rollup communication: All the rollups connect and communicate using the LxLy bridge, enabling seamless L2 to L2 bridging. - Rollup upgradability: All the deployed rollups should be upgradeable in accordance with its own governance mechanism. These functionalities are accomplished by modifying a few key components in the architecture of the zkEVM bridge version-1. zkEVM bridge version-1 Here is a brief review of the zkEVM bridge's architecture. Version-1 consists mainly of three (3) smart contracts: - The bridge contract (PolygonZkEVMBridge.sol), which handles transfer of assets and messages between networks. - The global exit root manager contract (PolygonZkEVMGlobalExitRoot.sol), which facilitates synchronization of state-info between the L2 and the L1. - The Polygon zkEVM consensus contract (PolygonZkEVMEtrog.sol), which handles the sequencing and verification of transactions in the form of batches. Global exit trees review Critical to the design of the LxLy bridge are exit trees and the global exit tree. Each chain has a Merkle tree called an exit tree, to which a leaf containing data of each asset-transfer is appended. Since such a leaf records data of the asset exiting the chain, it is called an Exit Leaf. Another Merkle tree whose leaves are roots of the various exit trees is formed, and it is called the global exit tree. The root of the global exit tree is the single source of state-truth communicated between rollups. It is the global exit root manager contract's responsibility to update the global exit tree root and acts as a custodian for the global exit tree's history. A complete transfer of assets in version-1 involves invoking three smart contracts; PolygonZkEVMEtrog.sol, PolygonZkEVMBridge.sol and PolygonZkEVMGlobalExitRoot.sol. The below figure depicts a bridge of assets and a claim of assets; !Figure 1: An asset transfer and three Smart Contracts Observe, in the above figure, that the consensus contract (PolygonZkEVMEtrog.sol) is able to: - Retrieve the global exit root from the mainnet, and make it available in L2 - Update the exit tree root in the global exit tree root manager. LxLy bridge version-2 design Multiple ZK rollups such as zkEVMs, zk-Validiums or zk-VMs, can be created and connected through the same LxLy bridge. Thanks to the introduction of an additional smart contract called the rollup manager. Although the logic for bridging or claiming assets from one network to another remains the same, individual L2 consensus contracts no longer handle verification of their own network's sequenced batches. In the context of the LxLy bridge, the rollup manager contract verifies sequenced batches from various networks. In addition to verification of sequenced batches, the rollup manager contract also creates consensus contracts for networks connecting via the LxLy bridge. What remains unchanged from version-1? The strategy to separate the bridge logic from the global exit root logic remains intact. This is key to achieving interoperability. Consensus contracts of each connected network handle the sequencing of their own batches, but send the batch data to the rollup manager contract for verification. The rollup manager contract stores the information of the sequenced batches in the form of an accumulated input hash, as in the version-1 of the zkEVM bridge. Once sequenced batches have been verified, the global exit tree gets updated, in an approach similar to the zkEVM bridge version-1. !Figure 2: New version of bridge Rollup manager's role The rollup manager is in charge of the following lists of availability: - Rollup consensus mechanisms: The list may consist of consensus contracts such as PolygonZkEVMEtrog.sol or zkValidium.sol. - Verifier contracts: For example, the PolygonZkEVMEtrog.sol uses the Verifier.sol contract for verification of batches. The governance contract oversees consensus mechanisms and verifiers that can be added to the respective lists. The rollup manager contract has the relevant function for adding a new rollup: bash function createNewRollup(uint32 rollupTypeID, uint64 chainID, address admin, address sequencer, address gasTokenAddress, uint32 gasTokenNetwork, string memory sequencerURL, string memory networkName) external onlyRole(CREATEROLLUPROLE) { //

Check that rollup type exists if (rollupTypeID == 0 || rollupTypeID > rollupTypeCount) { revert RollupTypeDoesNotExist(); } - To create and connect a rollup to the LxLy bridge, - The developer selects the consensus and verifier for the required rollup amongst those available in the rollup manager's lists, - Requests creation of a rollup with the selected specifications, - Governance contract invokes the rollup manager's addNewRollupType() function, - Once a rollup is created, the transfer of assets can be processed in the usual manner. Overall flow of events The following diagram captures the following flow of events, most of which are handled by the rollup manager contract: - Updating rollup manager's lists. - Creating rollups. - Sequencing of batches. - Aggregation or proving of batches. - Verification of batches. - Updating the global exit root. !Figure 3: Events flow related to RollupManager.sol Conclusion Although the LxLy bridge is still in development, it is a central component to the AggLayer which offers multi-chain interoperability. The LxLy bridge currently works with the Polygon zkEVM as the L2 and the Ethereum network as L1. The next step is to enable developers wishing to create a zk-rollup to choose between a zkEVM and a zkValidium rollup. The idea of handling verification of several networks in a single contract, is a pre-cursor to the ultimate and envisaged interop layer for the Polygon ecosystem. The code for the LxLy bridge version-2 can be found here. # ulxly-exit-trees-node.md: This section provides details of how the mainnet global exit tree and the rollups' global exit trees are constructed. New global exit tree Due to the presence of multiple layers in the uLxLy bridge, it becomes necessary to adjust the global exit tree so as to accommodate exits across all these layers. The mainnet has a local exit tree built as an append-only tree of 32 levels. Each rollup has a local exit tree which is also built as an append-only tree of 32 levels. All the rollups are grouped together in a tree of rollups called the rollup exit tree, which is again built as an append-only tree of 32 levels. The figure below illustrates how the global exit tree has been modified in order to introduce more than one rollup. The design of the updated global exit tree has two main branches: - One containing the root of the mainnet exit tree, and - The other branch containing the root of the rollup exit tree, which summarizes exit trees of all the rollups. The rollup exit tree has as its leaves all the local exit roots of the different rollups. !ulxly-mainnet-and-rollup-exit-trees Rollup identifiers Every rollup has a set of distinct identifiers that are essential for its functioning and interaction within the larger network ecosystem. - The $\texttt{chainID}$ is a unique identifier that distinguishes a rollup from other chains in the Ethereum ecosystem, and it is crucial for preventing replay attacks. See the list of chain IDs of different networks here. - A $\texttt{networkID}$ identifier defines a rollup in the Polygon ecosystem, allowing network participants to uniquely identify and interact with the rollup. The Ethereum mainnet is identified by $\texttt{networkID = 0}$, while the $\texttt{networkID = 1}$ is reserved for the Polygon zkEVM, and so on. - The $\texttt{rollupIndex}$ is an identifier used to identify a rollup within the rollup tree. The first rollup, being the Polygon zkEVM, has $\texttt{rollupIndex = 0}$. And in general, $\texttt{rollupIndex = networkID - 1}$. Global index When creating and verifying proofs, an index called $\texttt{globalIndex}$ is used to uniquely locate a leaf in the new global exit tree. A $\texttt{globalIndex}$â€‹â€‹ is a 256-bit string composed of unused bits, mainnet flag, rollup index bits, and local root index bits. $$ \texttt{globalIndex} = \texttt{(unused bits)} \texttt{(mainnet flag)} \texttt{(rollupIndex)} \texttt{(local root index)} $$ Starting from the most significant bit, a $\texttt{globalIndex}$ consists of the following bits: - 191 bits of unused bits: These bits are unused, and can be filled with any value. The best option is to fill them with zeros because zeros are cheaper. - 1 bit of mainnet flag: This single bit serves as a flag indicating whether an exit pertains to a rollup (represented by 0) or the mainnet (indicated by 1). - 32 bit of the rollup Index: These bits indicate the specific rollup being pointed at, within the rollup exit tree. These bits are therefore only used whenever mainnet flag is 0. - 32 bits of the local root index: These bits indicate the specific index being pointed at, within each rollup's local exit tree. The figure below depicts how the $\texttt{globalIndex}$ is interpreted: - The mainnet exit tree has the $\texttt{globalIndex = X1X3}$, where - the $\texttt{mainnet flag}$ equals $\texttt{1}$ to indicates that it's the mainnet, and - the $\texttt{local root index}$ being $\texttt{3}$ points at the fourth leaf in the mainnet exit tree. - The rollup 9002's exit tree has the $\texttt{globalIndex = X012}$, where - the $\texttt{mainnet flag}$ is $\texttt{0}$ indicating that it's a rollup, - the $\texttt{rollupIndex = 1}$ means the rollup is the second in the rollup exit tree, and - the $\texttt{local root index = 3}$, pointing at the fourth leaf in rollup 9002's exit tree. !ulxly-mainnet-and-rollups-exit-trees-2 # ulxly-interchain-exchanges.md: This covers node configuration parameters, interchange of tokens among chains, and the mention of an issue with CREATE2. Node configuration The node configuration for a rollup (or validium) network typically involves specifying various parameters and addresses that are crucial for its operation. The figure below provides a detailed breakdown of network parameters, which are given here as a snippet of node configuration within a TOML file. !ulxly-toml-file-node-config The $\texttt{chainId}$ is the chain identifier of the base layer (i.e., The Ethereum mainnet in this case). The $\texttt{genesisBlockNumber}$ is the L1 block number in which the rollup or validium is created. Gas tokens and inter-layer exchanges The native currency for paying gas at a certain layer can be: - Any ERC-20 token instance on any other layer, or - L1 ETH. If a token is used for paying gas in a network, it is referred to as the gas token for the layer. If we are using a gas token at a layer, it is still possible to send L1 ETH to the layer. In this case, the ETH gets accounted for in an ERC-20 contract called wrapped ETH (or W-ETH), which is just another ERC-20 instance. Note that W-ETH is different from the contract W-ETH (a contract for converting ETH into an ERC-20 token that runs on L1 at this address $\text{0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2}$.) Regarding the creation of the ERC-20 tokens with CREATE2: - Use $\texttt{salt = 0}$ to create the W-ETH contract. - Use $\texttt{salt = tokenInfoHash}$ for the rest of the wrapped tokens of the layer with tokenInfoHash defined as the following hash: $$ \texttt{tokenInfoHash = keccak256(originNetwork, originTokenAddress)} $$ As a final remark, note that L1 ETH is the only native currency that can be used as a native currency in another layer. The figure below, illustrates various scenarios of inter-layer exchanges that can occur within the system. With focus on LY as a layer of interest, the diagram depicts several scenarios, such as - Bridging an ERC-20 token from mainnet to LY - Bridging L1 ETH to LY gas token, or - Bridging a wrapped ERC-20 token living on LX to LY ETH. !ulxly-mainnet-lx-ly-bridge Upgradable CREATE2 factory issue Note that the bridge contract is a factory of ERC-20 token instances created with $\texttt{CREATE2}$. Recall that $\texttt{CREATE2}$ uses the following formula to compute the address of the instances: $$ \mathtt{instance\address} = \mathtt{hash(0xFF, sender, salt, creationBytecode, [args])} $$ Recall also that in the bridge contract, the mapping $\texttt{tokenInfoToWrappedToken}$ stores the addresses of all the wrapped ERC-20 tokens of the layer. The problem is that if we change the $\texttt{creationBytecode}$ of the ERC-20 token contract, this will change all the addresses of the contract instances and breaks the data of the mapping. The $\texttt{creationBytecode}$ will change with high probability if we compile the factory (in our case the bridge) with another version of the Solidity compiler. In this case, we had two options: (a) Freeze the Solidity compiler version for the development of the whole bridge contract. (b) Freeze the $\texttt{creationBytecode}$ of the ERC20 token contract. We opted for the second solution because the ERC20 code is not prone to change so much, while freezing the compiler (and the language) for the whole bridge could constrain its future development. Taking this approach, in the $\mathtt{BASE\INIT\BYTECODE\WRAPPED\TOKEN}$ variable of the bridge contract you can find the pre-compiled $\texttt{creationBytecode}$ of our ERC20 token contract. # ulxly-rollupmanager.md: Unified bridge is a specific instance of the LxLy bridge that allows several chains to connect to it. It is designed for adding chains to the Polygon ecosystem, and thus allow interoperability of many chains, both L2 and L1 networks. It aims at streamlining the creation and management of different L2 layers within the Polygon network. This includes rollups and validiums among the Polygon network, ensuring possible exchanges of messages among them. !!!caution While it is technically imprecise, for the sake of simplicity, we will henceforth refer to both rollups and validiums as 'rollups'. The rollup manager In order to achieve the afore-mentioned goal, a new smart contract called $\texttt{RollupManager}$ has been written for the purposes of managing creation of rollups and verification of their batches. Creation of a rollup could mean one of two things: - Creating and initializing a brand new rollup. - Incorporating an already existing rollup under the management of the $\texttt{RollupManager}$. Creating new rollups The first scenario involves newly created rollups, which have not yet been initialized, and therefore have an empty state. When a user triggers the rollup manager's function to create a new rollup, the $\texttt{RollupManager}$ should - Populate the configuration parameters, - Initialize the rollup by generating and writing the genesis block, - Sequence transactions for initializing the bridge contract attached to the rollup. The figure below depicts the process of creating and initializing a new rollup instance. Observe that the state tree is empty in this situation. !ulxly-process-initializing-new-rollup Incorporating existing rollups When an operational rollup is already present on the Ethereum network, a user with the necessary rights can incorporate it into the Rollup Manager for centralized management. In this scenario, the consensus needs no initialization because the rollup, its genesis block, and corresponding Bridge, have already been established. The diagram below depicts integration of an existing and operational rollup into the $\texttt{RollupManager}$, and showcases the process which needs no initialization because the rollup is already established. In this case, the state tree has information, and this is simply added to the manager. !ulxly-existing-rollup-incorporate Rollup types Each new rollup has a $\texttt{RollupType}$ attribute attached to it. And it specifies the following parameters: - The consensus implementation address, which is the address of the contract responsible for sequencing batches. - The verifier address which implements the $\texttt{IVerifierRollup}$ interface. The interface enables verification of each proof sent by the aggregator. - The $\texttt{forkID}$, for tracking changes in the rollup processing. - A rollup compatibility identifier, which is used to prevent compatibility errors when the rollup needs an upgrade. - The obsolete flag, which is a flag for indicating whether the rollup is obsolete or not. - The genesis block, which is the rollup's initial block and can include a small initial state. Note that several rollups can be of the same $\texttt{RollupType}$â€‹, which means they all share consensus and batch verification smart contracts. The $\texttt{RollupManager}$ contract has functions; $\texttt{addNewRollupType()}$ and $\texttt{obsoleteRollupType()}$ for adding or obsoleting a rollup type. It is not possible to create a new rollup of an obsolete rollup type. Rollup data Although several rollups can be of the same $\texttt{RollupType}$, it's important for each rollup store its state data. This state data is included in a structure called $\texttt{RollupData\{\}}$. The $\texttt{RollupData\{\}}$ struct contains; - Information about the current state of the rollup (e.g., The current batch being sequenced or verified, the states root for each batch, etc.) - Information about the bridge within the rollup, such as the current local exit root. - Data about forced batches, which is documented here in the Polygon Knowledge Layer. !ulxly-rollupdata-structure-definition Creating a rollup Each rollup is associated with either a single rollup type or none. In order to create a rollup of a certain rollup type, we can use the $\texttt{createNewRollup()}$ function by specifying: - The associated non-obsolete rollup type identifier, which should be existing. - The $\texttt{chainID}$ of the rollup, which should be new among the Polygon network's rollup chain IDs. - The address of the rollup admin, who is able to update several parameters of the consensus contract. For instance, setting a trusted sequencer or a force batches address. - The address of the trusted sequencer, which is the node responsible for sending the transaction for executing the $\texttt{sequenceBatches()}$ function. - The address of the token address used to pay gas fees, in the newly created rollup. When creating a new rollup, OpenZeppelin's transparent proxy pattern is employed by generating an instance of the $\texttt{PolygonTransparentProxy}$ contract. During this process, the consensus contract is specified by the rollup type serving as its implementation. Since the rollup is currently not initialized, the $\texttt{RollupData}$ is partially filled, and stored in the $\texttt{rollupIDToRollupData}$â€‹ mapping within the contractâ€™s storage. The rollup creation process is concluded by calling the $\texttt{initialize()}$â€‹ function of the consensus, which is in charge of setting previously specified addresses in the consensus contract. Below is a schematic representation of the transparent proxy pattern within the Rollup Manager context. !ulxly-transparent-proxy-pattern Proxies are frequently utilized in Ethereum for upgradability, and the exact usage of proxies in the Polygon zkEVM's upgradability is discussed here. # ulxly-sequence-verify.md: This section entails the flows for sequencing and verification of proofs. Sequencing flow The sequencing flow starts with the sequencer invoking the $\texttt{sequenceBatches()}$ function, which is within the consensus contract, to send batches that are to be sequenced. Since state information must be stored within the $\texttt{RollupManager}$ contract, a callback function called $\texttt{onSequenceBatches()}$ is triggered to store this data in the corresponding $\texttt{RollupData}$ struct. The figure below depicts a simplified sequencing flow within the rollup manager component: - It starts when the sequencer calls the $\texttt{sequenceBatches()}$ function, - Which in turn invokes a callback function $\texttt{onSequenceBatches()}$, - Followed by the rollup manager storing the sequence data in the $\texttt{RollupData}$ struct. !ulxly-sequencer-consensus-rollupmanager Verification flow Recall that the aggregator is responsible for constructing proofs that validate correct processing of batches. Once a proof is constructed, the Aggregator transmits it to the $\texttt{RollupManager}$ for verification, by invoking the $\texttt{verifyBatches()}$ function. The $\texttt{RollupManager}$ calls the $\texttt{verifyProof()}$ function in the verifier contract, which either validates the proof or reverts if the proof is invalid. If verification of the proof is successful, a callback function $\texttt{onVerifyBatches()}$ in the consensus contract is called. The $\texttt{onVerifyBatches()}$ function emits the $\texttt{VerifyBatches}$ event, containing important details of the processed batches, such as the last verified batch. The figure below depicts the verification flow within the rollup manager component: - It starts when the aggregator calls $\texttt{verifyBatches()}$ function. - Then the rollup manager invokes the $\texttt{verifyProof()}$ function, which involves a secondary stateless $\texttt{Verifier}$ contract. - Successful verification of a proof is followed by a call to the $\texttt{onVerifyBatches()}$ function. - At the end of the process, the consensus contract emits the $\texttt{verifyBatches}$ event. !ulxly-consensus-rollupmanager-aggregator # ulxly-updating-rollups.md: !!!caution While it is technically imprecise, for the sake of simplicity, we refer to both rollups and validiums as 'rollups'. Updating a rollup It is often necessary to enable upgradeability of rollups. More specifically, a user with appropriate rights can change the consensus implementation and the type of a certain rollup. Such a user can therefore modify the sequencing or verification procedures of a rollup. In order to change the consensus, the function $\texttt{UpdateRollup()}$ needs to change the transparent proxy implementation. In the upgrading procedure, the $\texttt{rollupCompatibilityID}$ comes into play: - In order to avoid errors, we can only upgrade to a rollup type having the same compatibility identifier as the original one. If this is not the case, the transaction is reverted, raising the $\texttt{UpdateNotCompatible}$ error. Adding existing rollups Rollups that are already deployed and working, do not follow any rollup type. Such an existing rollup can be added to the $\texttt{RollupManager}$ via the $\texttt{addExistingRollup}()$ function, by specifying its current address. When the verifier implements the $\texttt{IVerifierRollup}$ interface, it requests only for the raw consensus contract address, as it will not be used directly but through a proxy to allow upgradeability options. As mentioned before, rollups that are deployed and already in operation can be added to the $\texttt{RollupManager}$ in order to allow unified management. In this case, the $\texttt{addExistingRollup()}$ function is called. Since the rollup has previously been initialized, the following information needs to be provided: - The consensus contract, implementing the $\texttt{IPolygonRollupBase}$ interface. - The verifier contract, implementing the $\texttt{IVerifierRollup}$ interface. - The $\texttt{forkID}$ of the existent rollup. - The $\texttt{chainID}$ of the existent rollup. - The genesis block of the rollup. - The $\texttt{rollupCompatibilityID}$. Observe that most of these parameters were actually provided by the $\texttt{RollupType}$, but $\texttt{RollupData}$ of already existing rollups is constructed by hand, since they do not follow any rollup type as yet. # index.md: zkNode is the software needed to run a zkEVM node. It is a client that network users require to synchronize and know the state of the Polygon zkEVM. The main actors influencing the L2 State and its finality are the trusted Sequencer and trusted Aggregator. The zkNode architecture is modular in nature. See the below diagram for more clarity. !zkNode Diagram Most important to understand, is the primary path taken by transactions; from when users submit the transactions to the zkEVM network up until they are finalized and incorporated in the L1 State. Polygon zkEVM achieves this by utilizing several actors. Here is a list of the most prominent zkEVM components; - The Users, who connect to the zkEVM network by means of an RPC node (e.g., MetaMask), submit their transactions to a database called Pool DB. - The Pool DB is the storage for transactions submitted by Users. These are kept in the pool waiting to be put in a batch by the Sequencer. - The Sequencer is a node responsible for fetching transactions from Pool DB, checking if the transactions are valid, then putting valid ones into a batch. The sequencer submits all batches to the L1 and then sequences the batches. By doing so, the sequenced batches should be included in the L1 State. - The Synchronizer is the component that updates the State DB by fetching data from Ethereum through Etherman. - The Etherman is a low-level component that implements methods for all interactions with the L1 network and smart contracts. - The State DB is a database for permanently storing state data (but not the Merkle trees). - The Aggregator is another node whose role is to produce proofs attesting to the integrity of the Sequencer's proposed state change. These proofs are zero-knowledge proofs (or ZK-proofs) and the Aggregator employs a cryptographic component called the Prover for this purpose. - The Prover is a complex cryptographic tool capable of producing ZK-proofs of hundreds of batches, and aggregating these into a single ZK-proof which is published as the validity proof. Users can set up their own local zkNode by following this guide here, or a production zkNode as detailed here. zkNode roles The zkNode software is designed to support execution of multiple roles. Each role requires different services to work. Although most of the services can run in different instances, the JSON RPC can run in many instances (all the other services must have a single instance). RPC endpoints Any user can participate in this role, as an RPC node. Required services and components: - JSON RPC: can run on a separated instance, and can have multiple instances. - Synchronizer: single instance that can run on a separate instance. - Executor & Merkletree: service that can run on a separate instance. - State DB: Postgres SQL that can run on a separate instance. There must be only one synchronizer, and it's recommended that it must have exclusive access to an executor instance, though not necessarily. The synchronizer role can run perfectly in a single instance, but the JSON RPC and executor services can benefit from running in multiple instances, if the performance decreases due to the number of received requests. - zkEVM RPC endpoints - zkEVM RPC Custom endpoints documentation Trusted sequencer This role can only be performed by a single entity. This is enforced in the smart contract, as the related methods of the trusted sequencer can only be performed by the owner of a particular private key. Required services and components: - JSON RPC: can run on a separated instance, and can have multiple instances. - Sequencer & synchronizer: single instance that needs to run them together. - Executor & Merkletree: service that can run on a separate instance. - Pool DB: Postgres SQL that can run on a separate instance. - State DB: Postgres SQL that can run on a separate instance. Note that the JSON RPC is required to receive transactions. It's recommended that the JSON RPC runs on separated instances, and potentially more than one (depending on the load of the network). It's also recommended that the JSON RPC and the sequencer don't share the same executor instance, to make sure that the sequencer has exclusive access to an executor Aggregator This role can be performed by anyone. Required services and components: - Synchronizer: single instance that can run on a separate instance. - Executor & Merkletree: service that can run on a separate instance. - State DB: Postgres SQL that can be run on a separate instance. - Aggregator: single instance that can run on a separate instance. - Prover: single instance that can run on a separate instance. - Executor: single instance that can run on a separate instance. It's recommended that the prover is run on a separate instance, as it has important hardware requirements. On the other hand, all the other components can run on a single instance. # arithmetic-sm.md: The Arithmetic state machine is a secondary state machine that also has an executor (the Arithmetic SM executor) and an internal Arithmetic program (a set of verification rules written in the PIL language). The Arithmetic SM executor is available in two languages: Javascript and C/C++. It is one of the six secondary state machines receiving instructions from the Main SM executor. The main purpose of the Arithmetic SM is to carry out elliptic curve arithmetic operations, such as Point Addition and Point Doubling as well as performing 256-bit operations like addition, product or division. Standard elliptic curve arithmetic Consider an elliptic curve E defined by $y^2 = x^3 + ax + b$ over the finite field $\mathbb{F} = \mathbb{Z}_p$, where p is the prime, $$ p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 -2^6 - 2^4 - 1 $$ Set the coefficients $a = 0$ and $b = 7$, so that E reduces to $$ y^2 = x^3 + 7 $$ Point addition Given two points, $P = (x1,y1)$ and $Q = (x2,y2)$, on the curve E with $x1 \neq x2$, the point $P+Q = (x3,y3)$ is computed as follows, $$ x3 = s^2 - x1 - x2,\quad \\ y3 = s (x1 - x3) - y1 $$ Point doubling Given a point $P = (x1,y1)$ on the curve E such that $P \neq \mathcal{O}$, the point $P+P = 2P = (x3,y3)$ is computed as follows, $$ x3 = s^2 - 2x1, \qquad\quad \\ y3 = s (x1 - x3) - y1, $$ where, $$ s = \dfrac{3x1^2}{2y1}. $$ Field arithmetic Several 256-bit operations can be expressed in the following form: $$ A \cdot B + C = D \cdot 2^{256} + E \tag{\bf{Eqn A}} $$ where A, B, C, D and E are 256-bit integers. For instance, if $C = 0$, then $\bf{Eqn\ A}$ states that the result of multiplying A and B is E with a carry of D. That is, D is the chunk that exceeds 256 bits. Or, if $B = 1$, $\bf{Eqn\ A}$ states that the result of adding A and C is the same as before: E with a carry of D. Similarly, division and modular reductions can also be expressed as derivatives of $\bf{Eqn\ A}$.

These operations are performed in the Arithmetic state machine, with registers satisfying the following PIL relation, $$ \text{EQ}0 \colon \quad x1 \cdot y1 + x2 - y2 \cdot 2^{256} - y3 = 0 $$ Remark Since the above elliptic curve operations are implemented in the PIL language, it is more convenient to express them in terms of the constraints they must satisfy. These constraints are: $$ \begin{aligned} \text{EQ}0 \colon \quad &x1 \cdot y1 + x2 - y2 \cdot 2^{256} - y3 = 0, \\ \text{EQ}1 \colon \quad &s \cdot x2 - s \cdot x1 -y2 + y1 + q0 \cdot p = 0, \\ \text{EQ}2 \colon \quad & 2 \cdot s \cdot y1 - 3 \cdot x1 \cdot x1 + q0 \cdot p = 0, \\ \text{EQ}3 \colon \quad & s \cdot s - x1 - x2 - x3 + q1 \cdot p = 0, \\ \text{EQ}4 \colon \quad & s \cdot x1 - s \cdot x3 - y1 - y3 + q2 \cdot p = 0, \end{aligned} $$ where $q0,q1,q2 \in \mathbb{Z}$, implying that these equations hold true over the integers. This approach is taken because of the need to compute divisions by p. Note that only three possible computation scenarios can arise: 1. $\text{EQ}0$ is activated while the rest are deactivated, 2. $\text{EQ}1$, $\text{EQ}3$ and $\text{EQ}4$ are activated but $\text{EQ}0$ and $\text{EQ}2$ are deactivated, 3. $\text{EQ}2$, $\text{EQ}3$ and $\text{EQ}4$ are activated and $\text{EQ}0$ and $\text{EQ}1$ are deactivated. Since at most, one of $\text{EQ}1$ and $\text{EQ}2$ are activated in any scenario, we can afford "sharing" the same $q0$ for both. Motivated by the implemented operations, the Arithmetic SM is composed of 6 registers, each of which is composed of 16 sub-registers of 16-bit (2 byte) capacity, adding up to a total of 256 bits per register. $$ \begin{aligned} x1,\ y1,\ x2,\ y2,\ x3,\ y3. \end{aligned} $$ There is also a need to provide s and $q0,q1,q2$, which are also 256-bit field elements. How operations are performed Compute the previous operations at 2-byte level. For instance, if one is performing the multiplication of $x1$ and $y1$, at the first clock $x1[0] \cdot y1[0]$ is computed. Then, $(x1[0] \cdot y1[1]) + (x1[1] \cdot y1[0])$ is computed in the second clock, followed by $(x1[0] \cdot y1[2]) + (x1[1] \cdot y1[1]) + (x1[2] \cdot y1[0])$ in the third, and so on. As depicted in the below figure, this process is completely analogous to the schoolbook multiplication. However, it is performed at 2-byte level, instead of decimal level. !School Multiplication Example Use the following notation: $$ \begin{aligned} \mathtt{eq\ } &= x1[0] \cdot y1[0] \\ \mathtt{eq'} &= x1[0] \cdot y1[1] + x1[1] \cdot y1[0] \end{aligned} $$ But then, the carry generated by $\mathtt{eq}$ has to be taken into account by $\mathtt{eq'}$. Going back to our equations $\text{EQ}0, \text{EQ}1, \text{EQ}2, \text{EQ}4$; let's see how the operation is performed in $\text{EQ}0$. 1. Compute $\mathtt{eq}0 = (x1[0] \cdot y1[0]) + x2[0] - y3[0]$ 2. Compute $\mathtt{eq}1 = (x1[0] \cdot y1[1]) + (x1[1] \cdot y1[0]) + x2[1] - y3[1]$ 3. $\mathtt{eq}2 = (x1[0] \cdot y1[2]) + (x1[1] \cdot y1[1]) + (x1[2] \cdot y1[0]) + x2[2] - y3[2]$ This is continued until one reaches the computation, $\mathtt{eq}{15} = (x1[0] \cdot y1[15]) + (x1[1] \cdot y1[14]) + \dots + x2[15] - y3[15]$. At this stage $y2$ comes into place. Since the first 256 bits of the result of the operation have been filled (and the result can be made of more than 256-bits), a new register is needed to store the rest of the output. We change the addition of $x2[i] - y3[i]$ by $-y2[i]$. Therefore, we obtain that: $$ \begin{aligned} \mathtt{eq}{16} &= (x1[1] \cdot y1[15]) + (x1[2] \cdot y1[14]) + \dots - y2[0], \\ \mathtt{eq}{17} &= (x1[2] \cdot y1[15]) + (x1[3] \cdot y1[14]) + \dots - y2[1] \end{aligned} $$ and so on. Continuing until the last two: $$ \begin{aligned} \mathtt{eq}{30} &= (x1[15] \cdot y1[15]) - y2[14], \\ \mathtt{eq}{31} &= -y2[15]. \end{aligned} $$ Now, notice that the carry generated by the $\mathtt{eq}i$'s is not important for them. That means, if $\mathtt{eq}i = 10$, then what we really want the result to be is 0 and save 1 as a carry for the next operation. To express this fact as a constraint, we say that the following has to be satisfied: $$ \mathtt{eq} + \text{carry} = \text{carry}' \cdot 2^{16}, $$ where carry represents the carry taken into account in the actual clock, and carry' represents the carry generated by the actual operation. Remark A technicality is that carry is subdivided into two other $\text{carry}L$ and $\text{carry}H$ such that: $$ \text{carry} = \text{carry}L + \text{carry}H \cdot 2^{18}. $$ Source code The Polygon zkEVM repository is available on GitHub. Arithmetic SM executor: smarith folder Arithmetic SM PIL: arith.pil # binary-sm.md: The Binary state machine is one of the six secondary state machines receiving instructions from the Main SM executor. It is responsible for the execution of all binary operations in the zkProver. As a secondary state machine, the Binary state machine has the executor part, called the Binary SM executor, and an internal Binary PIL program, which is a set of verification rules written in the PIL language. The Binary SM executor is written in two languages: Javascript and C/C++. Binary operations on 256-bit strings The zkEVM performs the following binary operations on 256-bit strings: - The addition operation, denoted by ADD ($+$), adds two 256-bit numbers, - The subtraction operation, denoted by SUB ($-$), computes the difference between two 256-bit numbers, - The less-than operation, denoted by LT ($<$), checks if a 256-bit number is smaller than another 256-bit number, without considering the signs the numbers, - The signed less-than operation, denoted by SLT ($<$), checks if a 256-bit number is smaller than another 256-bit number, but takes into consideration the respective signs of the numbers, - The equal operation, denoted by EQ ($=$), checks if two 256-bit numbers are equal, - The AND operation, denoted by AND ($\land$), computes the bit-wise "AND" of two numbers, - The OR operation, denoted by OR ($\lor$), computes the bit-wise "OR" of two numbers, - The XOR operation, denoted by XOR ($\oplus$), computes the bit-wise "XOR" of two numbers, - The NOT operation, denoted by NOT ($\neg$), computes the bit-wise "NOT" of a binary number. Addition and subtraction operations In order to understand how the ADD and SUB operations work, one needs to first understand how the zkEVM encodes 256-bit strings to signed and unsigned integers. Below figure shows these codifications for 3-bit strings but the idea can be easily extended to 256-bit strings. !Figure 1: Codifications of 3-bit strings for signed and unsigned integers as used by the EVM Adding two strings is performed bit-by-bit using the corresponding carry. For example, add the 3-bit strings $\mathtt{0b001}$ and $\mathtt{0b101}$, where $\mathtt{0b}$ indicates that the number is binary; - Start with an initial $carry=0$ and add the least significant bits, $1+1+carry=1+1+0=0$, so the next carry becomes $carry'=1$. - Next, add the second least-significant bits using the previous carry, $0+0+carry = 0+0+1 = 1$, this time the next carry is $carry'=0$. - Finally, add the most significant bits, $0+1+carry=0+1+0=1$, with the final carry being $carry'=0$. - As a result: $\mathtt{0b001}+\mathtt{0b101} = \mathtt{0b110}$ with $carry=0$. The sum $\mathtt{0b001}+\mathtt{0b101} = \mathtt{0b110}$, for unsigned integers is $1+5=6$, while for signed integers encoded with two's complement, this sum is $1+(-3) =(-2)$. This is in line with Figure 1 above. In other words, the same binary sum can be done for both signed integers and for unsigned integers. Signed and unsigned 'Less Than' operations The operations LT and SLT behave differently from the ADD and SUB operations. When comparing unsigned integers using LT, the natural order for comparisons is applied. For example, $010 < 110$. That is, $2 < 6$. But when comparing signed integers using SLT, one must take into account the most significant bit that acts as the sign. - If the most-significant bits of the two strings being compared is the same, the natural order applies. For example, $101 < 110$. i.e., $-3 < -2$. Compare with Figure 1 above. - However, if the most significant bits of strings being compared are different, then the order must be flipped (bigger numbers start with 0). For example, $110 < 001$. i.e., $-2 < 1$. Finally, notice that with unsigned integers, there is a caveat since 4 and -4 have the same codification. AND, OR and XOR operations Consider the bit-wise operations; AND, OR, XOR and NOT. Bitwise means the operation is carried out bit-by-bit. As a result, there are no carry bits to be concerned about. This simplifies the implementation of the operations. The below table depicts the truth tables of AND, OR and XOR operators, respectively. !Truth Tables of bit-wise operations Notice that we do not consider the NOT operation. This is because the NOT operation can be easily implemented with the XOR operation, by taking an XOR of the given 256-bit string and $\texttt{0xff...ff}$. The design of Binary SM The executor of the Binary SM records the trace of each computation in the state machine, and this computational trace is used to prove correctness of computations. The execution trace is typically in the form of 256-bit strings. And the polynomial constraints, which every correct execution trace must satisfy, are described in a PIL file (or 'code'). For the Binary SM, these computations refer to the aforementioned binary operations, and use special codes for each of the operations. Codes for the binary operations Each operation that the Binary SM checks has an opcode as shown in the table below.

All Operations Checked by the Binary SM

$\textbf{Operation Name}$	$\textbf{Mnemonic}$	$\textbf{Symbol}$	$\textbf{BinOpCode}$
Addition	ADD	$+$	0
Subtraction	SUB	$-$	1
Less Than	LT	$<$	2
Signed Less Than	SLT	$<$	3
Equal To	EQ	$=$	4
Bitwise AND	AND	$\wedge$	5
Bitwise OR	OR	$\vee$	6
Bitwise XOR	XOR	$\oplus$	7
No Operation	NOP	NOP	$\star$

In instances where none of the defined binary operations is carried out, the Binary SM's operation is considered to be a NOP (No Operation), in which case any code not in the defined list of codes can be used. Internal byte plookups The Binary SM is internally designed to use Plookups of bytes for all the binary operations. It uses Plookups that contain all the possible input bytes and output byte combinations, $$ \text{byte}{in0} \star \text{byte}{in1} = \text{byte}{out}, $$ where $\star$ is one of the possible operations. When executing a binary operation between the 256-bit input strings, an execution trace is generated in cycles of 32 steps per operation. At each step, the corresponding byte-wise operation and any required extra information such as 'carries' or auxiliary values, form part of the computation trace. Additionally, each 256-bit string, the two inputs and the output, are expressed using 8 registers of 32-bits. Connection with the Main SM The constraint that connects the execution trace of the Main SM with the execution trace of the Binary SM is a Plookup, which is performed at each row of the Binary SM execution trace when the cycle is completed (this is when a register called $\texttt{RESET}$ is 1). The Plookup checks the operation code, the registries for the input and output 256-bit strings, and the final carry. Operating at byte-level This section provides examples of how the byte-wise operations work. A 256-bit integer $\mathbf{a}$ is herein denoted in vector form as $(a{31}, \dots, a1, a0)$ to indicate that, $$ \mathbf{a} = a{31}\cdot (2^8)^{31} + a{30}\cdot (2^8)^{30} + \cdots + a1\cdot2^8 + a0 = \sum{i = {31}}^{0} ai \cdot (2^8)^i, $$ where each ai is a byte that can take values between 0 and $2^8 - 1$. Example 1 If $\mathbf{a} = 29967$, its byte decomposition can be written as $\mathbf{a} = (\mathtt{0x75}, \mathtt{0x0F})$, because $\mathbf{a} = 29967 = 117 \cdot 2^8 + 15$, and in hexadecimal, $117 \mapsto \mathtt{0x75}$ and $15 \mapsto \mathtt{0x0F}$. Addition Here is how the addition operation on two 256-bit numbers is reduced to a byte-by-byte addition, and thus ready to use the byte-wise Plookup table. Observe that adding two bytes a and b (i.e., a and b are members of the set $[0, 2^8-1]$), may result in a sum c which cannot be expressed as a single byte. For example, if $a = \mathtt{0xFF}$ and $b = \mathtt{0x01}$, then, $$ a + b = \mathtt{0xFF} + \mathtt{0x01} = \mathtt{0x100}. $$ In byte-form, $c=\mathtt{0x00}$ and with $carry'=1$. This carry has to be taken care of when dealing with bytes. Consider now the process of adding two bytes. Example 2. Take for instance, $\mathbf{a} = (a1, a0) = (\mathtt{0xFF}, \mathtt{0x01})$ and $\mathbf{b} = (b1, b0) = (\mathtt{0xF0}, \mathtt{0xFF})$. - First add the less significant bytes: $$ \begin{aligned} a1 + b1 &= \mathtt{0x01} + \mathtt{0xFF} = c1 = \mathtt{0x00}, \\ carry1 &= 1. \end{aligned} $$ - Then, add the next significant byte, $$ \begin{aligned} a2 + b2 + carry1 &= \mathtt{0xFF} + \mathtt{0xF0} = c2 = \mathtt{0xF0}, \\ carry2 &= 1. \end{aligned} $$ The previous scheme depicts several cases that need to be treated separately; 1. If $a1 + b1 < 2^8$ and $a2 + b2 < 2^8$, then the sum $\mathbf{a} + \mathbf{b}$ is simply, $$ \mathbf{a} + \mathbf{b} = (a2 + b2, a1 + b1). $$ 2. If $a1 + b1 < 2^8$ but $a2 + b2 \geq 2^8$, then $a2 + b2$ does not fit in a single byte. Hence, the sum of $a2$ and $b2$ has to be written as $$ a2 + b2 = 1 \cdot 2^8 + c2, $$ â€‹for some byte $c2$. The addition $\mathbf{a} + \mathbf{b}$ is then computed as follows, $$ \mathbf{a} + \mathbf{b} = (1, c2, a1 + b1). $$ 3. If $a1 + b1 \geq 2^8$, then we have that: $$ a1 + b1 = 1 \cdot 2^8 + c1, $$ for some byte $c1$. Then we can write $$ \mathbf{a} + \mathbf{b} = (a2 + b2 + 1) \cdot 2^8 + c1. $$ Consider the following two scenarios: 1. If $a2 + b2 + 1 \geq 2^8$, then the sum takes the form: $$ a2 + b2 + 1 = 1 \cdot 2^8 + c2. $$ Therefore, the byte decomposition of $\mathbf{a} + \mathbf{b}$ is $$ \mathbf{a} + \mathbf{b} = (1, c2, c1). $$ 2. If $a2 + b2 + 1 < 2^8$, then the byte decomposition of $\mathbf{a} + \mathbf{b}$ is: $$ \mathbf{a} + \mathbf{b} = (c2, c1). $$ Observe that addition of 256-bit numbers can be reduced to additions at byte-level by operating through the previous cases in an iterative manner. Subtraction Reducing Subtraction to byte-level turns out to be trickier than Addition case. Suppose $\mathbf{a} = \mathtt{0x0101}$ and $\mathbf{b} = \mathtt{0x00FF}$. Observe that $\mathtt{0xFF}$ cannot be subtracted from $\mathtt{0x01}$ because $\mathtt{0xFF} > \mathtt{0x01}$. However, we know that the result is $\mathbf{a} - \mathbf{b} = \mathbf{c} = \mathtt{0x0002}$. In order to get this result, notice that the operation can be described as follows, $$ \begin{aligned} \mathbf{a} - \mathbf{b} & = (\mathtt{0x01} - \mathtt{0x00}) \cdot 2^8 + (\mathtt{0x01} - \mathtt{0xFF}) \\ & = (\mathtt{0x01} - \mathtt{0x00}) \cdot 2^8 - 2^8 + 2^8 + (\mathtt{0x01} - \mathtt{0xFF}) \\ & = (\mathtt{0x01} - \mathtt{0x00 - 0x01}) \cdot 2^8 + \mathtt{0xFF + 0x01} + \mathtt{0x01} - \mathtt{0xFF} \\ & = (\mathtt{0x00}) \cdot 2^8 + \mathtt{0x02} \end{aligned} $$ The output byte decomposition is $\mathbf{a} = (c1, c0) = (\mathtt{0x00}, \mathtt{0x02})$. Nonetheless, it may be necessary to look at more examples so as to better understand how subtraction works at byte-level in a more general sense. Consider now a subtraction of numbers with 3 bytes. Say, $a = \mathtt{0x0001FE}$ and $b = \mathtt{0xFEFFFF}$. First analyse the first two bytes, as in the previous example, $$ \begin{aligned} (\mathtt{0x01} - \mathtt{0xFF}) \cdot 2^8 + (\mathtt{0xFE} - \mathtt{0xFF}) &= (\mathtt{0x01} - \mathtt{0xFF} - \mathtt{0x01}) \cdot 2^8 + (\mathtt{2^8} + \mathtt{0xFE} - \mathtt{0xFF}) \\ & = (\mathtt{0x01} - \mathtt{0xFF - 0x01}) \cdot 2^8 + \mathtt{0xFF} \end{aligned} $$ But now observe that $\mathtt{0x01} - \mathtt{0xFF} - \mathtt{0x01}$ is also a negative value. Hence, there is a need to repeat the strategy and keep a carry to the next byte, $$ \begin{aligned} &(\mathtt{0x00} - \mathtt{0xFE}) \cdot 2^{16} + (\mathtt{0x01} - \mathtt{0xFF} - \mathtt{0x01}) \cdot 2^8 + \mathtt{0xFF} = \\ &(\mathtt{0x00} - \mathtt{0xFE} - \mathtt{0x01}) \cdot 2^{16} + (\mathtt{2^8} + \mathtt{0x01} - \mathtt{0xFF} - \mathtt{0x01}) \cdot 2^8 + \mathtt{0xFF} = \\ &(\mathtt{0x00} - \mathtt{0xFE} - \mathtt{0x01}) \cdot 2^{16} + \mathtt{0x01} \cdot 2^8 + \mathtt{0xFF}. \end{aligned} $$ Observe that the previous example is included in this case. In general, let $a = (ai)i$ and $b = (bi)i$, with ai, bi bytes, be the byte representations of a and b. Instead of checking if we can perform the subtraction $ai - bi$ for some bytes i, we are checking if $ai - bi - \texttt{carry} \geq 0$. Equivalently, we are checking if $ai - \texttt{carry} \geq bi$. The previous case can be recovered by setting $\texttt{carry} = 1$ and the first case corresponds to setting $\mathtt{carry = 0}$. We have two possible cases, - If $ai - \texttt{carry} \geq bi$, then $ai - bi - \texttt{carry}$ provides the corresponding i-th byte of the representation of $a - b$. - If $ai - \texttt{carry} < bi$ then we should compute the corresponding i-th byte of the representation of $a - b$ as, $$ 2^8 - bi + ai - \texttt{carry} = 255 - bi + ai - \texttt{carry} + 1. $$ However, we need to discuss the last step of our example. Observe that we cannot perform the operation $\mathtt{0x00} - \mathtt{0xFE} - \mathtt{0x01}$ since it corresponds to a negative value. But as we are working with unsigned integers, we do the two's complement and set the last byte to, $$ 2^8 - \mathtt{0xFE} + \mathtt{0x00} - \mathtt{0x01} = 255 - \mathtt{0xFE} + \mathtt{0x00} - \mathtt{0x01} + 1 = 255 - b3 + a3 - \texttt{carry} + 1. $$ Observe that this is also included in the case when $ai - \texttt{carry} < bi$, so we must not treat the last bit in a different manner. To end up with our example, we get the following byte representation of $a - b$, $$ c = (\mathtt{0x01}, \mathtt{0x01}, \mathtt{0xFF}) = \mathtt{0x01} \cdot 2^{16} + \mathtt{0x01} \cdot 2^8 + \mathtt{0xFF}. $$ Less than We want a byte-wise description of the less than comparator. For 256-bits integers, the operation $<$ outputs $c = 1$ if $a < b$ and $c = 0$ otherwise. As we are working in the natural integers order, the most significant byte decides and, if they are equal, we should consider the previous one until we can decide. Let us propose the example with $a = \mathtt{0xFF AE 09}$ and $b = \mathtt{0x FF AE 02}$. We know that $a > b$. Why? We should start at the most significant byte. We know that: $$ a \mathtt{>> 16} = \mathtt{0x FF} = \mathtt{0x FF} = b \mathtt{>> 16}. $$ Hence, we can not decide with this byte. And the same happens with the second byte, they are both equal to $\mathtt{0x AE}$. Hence, the less significant byte decides, $$ \mathtt{0x 09} > \mathtt{0x 02}. $$ However, the problem with our setup is that we must start with the less significant byte and climb up to the most significant byte. The strategy is to use some kind of a carry in order to "carry" the decisions from previous bytes. Let us do an example step by step, now with $a = \mathtt{0x FF AA 02}$ and $b = \mathtt{0x 01 AA 09}$. First of all, we compare the less significant bytes. Since: $$ \mathtt{0x 02} < \mathtt{0x 09}, $$ we set up $\mathtt{carry} = 1$. We then keep this decision until we finish to process all bytes or, alternatively, we should change to the complementary decision. Therefore, since the next two bytes are equal and we are not at the end, we maintain $\mathtt{carry}$ to 1. The previous step is the last one. We compare the most significant bytes, $$ \mathtt{0x FF} \not < \mathtt{0x 01}. $$ Henceforth, we should output a 0, independently to the previous carry decision. But, let us suppose now that $b = \mathtt{0x FF AA 09}$. Then, in this last step, we should output a 1, since $a < b$. The idea is that, in the last step, if both bytes are equal, we should output the decision carry $\mathtt{carry}$. In general, in the step i, comparing bytes ai and bi, we have 3 cases, - If $ai < bi$, we set $\mathtt{carry}$ to 1. If we are at the most significant byte, we output 1. - If $ai = bi$, we let $\mathtt{carry}$ unchanged in order to maintain the previous decision. If we are at the most significant byte, we output $\mathtt{carry}$. - If $ai > bi$, we set $\mathtt{carry}$ to 0. If we are at the most significant byte, we output 0. Signed less than In computer science, the most common method of representing signed integers on computers, is called two's complement. When the most significant bit is a one, the number is signed as negative. The way to express a negative integer x into two's complement form is chosen so that, among integers of the same sign, the lexicographical order is maintained. That is, if $a < b$ are signed integers of the same sign, then its two's complement representations preserve the same order. This is not true if the signs are different. For example, it is not surprising that: $$ 000\dots0 > 111\dots1 $$ using the two's complement encoding, because $111\dots1$ is negative and $000\dots0$ is positive. The two's complement form of negative integer x in a N-bits system is the binary representation of $2^N - x$. For example, let $x = -1$ and $N = 4$. Then, $$ 10000 - 0001 = 1111. $$ Hence, $-1 = 1111$ in this representation. It is easy to see that $-2 = 1110$ because: $$ 10000 - 0010 = 1110. $$ Hence, observe that $-1 > -2$ because $1111 > 1110$ and, conversely, the order is preserved for integers of the same sign. We describe a method to compare signed integers byte-wise. First of all, let us analyze the order among all the signed bytes, in order to understand how to compare them. Once we achieve this, the strategy is very similar to the previous Less Than. Let $a = (a{31}, a{30}, \dots, a0)$ and $b = (b{31}, b{30}, \dots, b0)$ be the byte-representation of the 256-bits unsigned integers a and b. We define $\texttt{sgn}(a) = a{31, 7}$, where $$ a{31} = \sum{i = 0}^7 a{31, i} \cdot 2^i $$ is the binary representation of $a{31}$. That is, $\texttt{sgn}(a)$ is the most significant bit of a or, equivalently, the "sign" of a. In a similar way, we define $\texttt{sgn}(b)$. Observe that it is easy to compare a and b if $\texttt{sgn}(a) \neq \texttt{sgn}(b)$. For example, $$ a = \mathtt{0b11111111} = \mathtt{0xFF} < \mathtt{0x00} = \mathtt{0b00000000} = b $$ because $\texttt{sgn}(a) > \texttt{sgn}(b)$ i.e. a is negative and b is positive. If $\texttt{sgn}(a) \neq \texttt{sgn}(b)$, we can simply compare a and b using the same strategy as before, because the natural lexicographic order is preserved in this case. Then, we have the following cases when comparing a and b: 1. If $\texttt{sgn}(a) = 1$ and $\texttt{sgn}(b) = 0$, then $a < b$. 2. If $\texttt{sgn}(a) = 0$ and $\texttt{sgn}(b) = 1$, then $a > b$. 3. If $\texttt{sgn}(a) = \texttt{sgn}(b)$, the order is the usual one and hence, we already know how to compare a and b. Recall that we are processing the bytes of a and b from the less significant bytes to the most significant bytes. Hence, we need to adapt our strategy following this order. The strategy is almost the same as in the unsigned operation. 1. First of all, we start comparing $a0$ and $b0$. (a) If $a0 < b0$, we set $\texttt{carry} = 1$. (b) Otherwise we set $\texttt{carry} = 0$. 2. For all $0 < i < 31$, we compare ai and bi. (a) If $ai < bi$, we set $\texttt{carry} = 1$. (b) If $ai = bi$, we leave $\texttt{carry}$ unchanged from the previous step. (c) Otherwise, we set $\texttt{carry} = 0$. 3. Now, we have to compare the last byte. We follow the described strategy of comparing the signs: (a) If $\texttt{sgn}(a) > \texttt{sgn}(b)$, we output a 1, so $a < b$. (b) If $\texttt{sgn}(a) < \texttt{sgn}(b)$, we output a 0, so $a < b$. (c) If $\texttt{sgn}(a) = \texttt{sgn}(b)$, we compare the last bytes $a{31}$ and $b{31}$ in the same way we have compared the previous bytes. We output 0 or 1 accordingly. 1. If $a{31} < b{31}$, we output a 1, so $a < b$. 2. If $a{31} = b{31}$, we output the previous $\texttt{carry}$, maintaining the last decision. 3. Otherwise, we output a 0, so $a \not < b$. Let us exemplify the previous procedure setting $a = \mathtt{0xFF FF FF 00}$ and $b = \mathtt{0x00 FF FF FF}$. We know that $a < b$, so we should output a 1. Observe that the less significant byte of a is leaser than the less significant byte of b. Hence, we should put $\texttt{carry}$ equal to 1. The next two bytes of a and b are both equal to $\mathtt{0xFF FF}$, therefore we maintain $\texttt{carry}$ unchanged equal to 1. However, since a is negative and b is positive, we should change the decision and output a 1, independently of the $\texttt{carry}$. Equality We want to describe the equality comparator byte-wise. For unsigned 256-bits integers, the operation $=$ outputs $c = 1$ if $a = b$ and $c = 0$ otherwise. This operation is very simple to describe byte-wise, since $a = b$ if and only if all its bytes coincide. Let us compare $a = \mathtt{0xFF 00 a0 10}$ and $b = \mathtt{0xFF 00 00 10}$ byte-wise. Observe that the first byte is the same $\mathtt{0x10}$, however the next byte is different $\mathtt{0xa0} \neq \mathtt{0x00}$. Hence, we can finish here and state that $a \neq b$. We describe an algorithm in order to proceed processing all the bytes. We use a carry to mark up when a difference among bytes has $\textbf{not}$ been found (i.e. if $\texttt{carry}$ reach 0, then a and b should differ). Hence, the algorithm to compare two 32-bytes integers $a = (a{31}, a{30}, \dots, a{0})$ and $b = (b{31}, b{30}, \dots, b0)$ is the following: 1. First of all, since no differences have been found up to this point, set $\texttt{carry}$ equal to 1. 2. Now, compare $a0$ and $b0$, (a) If $a0$ and $b0$ are equal, then leave $\texttt{carry}$ unchanged equal to 1. (b) If $a0 \neq b0$, then set $\texttt{carry}$ equal to 0, which implies that $a \neq b$. 3. When comparing bytes ai and bi for $0 < i \leq 31$. (a) If $ai = bi \textbf{ and } \texttt{carry} = 1$, we should leave $\texttt{carry}$ unchanged and, if $i = 31$, we should output a 1 because $a = b$. The reason of demanding $\texttt{carry} = 1$ in the enter condition is because we should ensure that, if $\texttt{carry} = 0$ in a previous step, we must never enter to this block and change the non-equality decision. This is because if $ai \neq bi$ for some i, then $a \neq b$. (b) Hence, if $ai \neq bi$, we should set $\texttt{carry} = 0$ and output a 0 if $i = 31$. Bitwise operations We describe all bitwise operations at once because they are the easiest ones, since we do not need to introduce carries. Now, the idea is to extend this operation bitwise. That is, if we have the following binary representations of $a = (a{31}, a{30}, \dots, a{0})$ and $a = (b{31}, b{30}, \dots, b{0})$ where $ai, bi \in \{0, 1\}$, then we define, $$ a \star b = (ai \star bi)i = (a{31} \star b{31}, a{30} \star b{30}, \dots, a0 \star b0) $$ for $\star$ being $\land, \lor$ or $\oplus$. For example, if $a = \mathtt{0xCB} = \mathtt{0b11001011}$ and $b = \mathtt{0xEA} = \mathtt{0b11101010}$ then, $$ \begin{aligned} a \land b &= \mathtt{0b11001010} = \mathtt{0xCA},\\ a \lor b &= \mathtt{0b11101011} = \mathtt{0xEB},\\ a \oplus b &= \mathtt{0b00100001} = \mathtt{0x21}. \end{aligned} $$ Binary SM in summary The Binary SM has 8 registries, each with a 32-bit Input/Output capacity. i.e., a total of 256 bits. It carries out binary computations in accordance with instructions from the Main SM executor. The binary operations it executes, together with their specific opcodes, are: 1. The common operations; the No-Operation NOP, Addition ADD and Subtraction SUB. Their corresponding special opcodes are; 0, 1 and 2, respectively. 2. The Boolean operations; Less Than LT, Greater Than GT, Signed Less Than SLT, $\text{Signed Greater Than }$ SGT, Equal EQ and Is-Zero ISZERO. Their special opcodes are, 3, 4, 5, 6 and 7, respectively. 3. The logical operations; AND, OR, XOR and NOT, each with its special opcode; 9 ,10, 11 and 12, respectively. Firstly, the Binary SM executor translates the Binary Actions into the PIL language. Secondly, it executes the Binary Actions. And thirdly, it uses the Binary PIL program, to check correct execution of the Binary Actions using Plookup. Translation to PIL language It builds the constant polynomials, which are generated once-off at the beginning. These are; - the 4 bits long operation code POPCODE, - the 1-bit Carry-in PCIN, - the Last-byte PLAST, - the 1 byte input polynomials PA and PB, - the 16-bit output polynomial PC, - the 1-bit Carry-out PCOUT. It also creates constants required in the Binary PIL program; - RESET is used to reset registry values every time the state machine completes a cycle of state transitions, - FACTOR, which is an array of size 8, is used for correct placement of output registry values. Execution of Binary actions The crux of the Binary SM executor is in the lines 371 to 636 of smbinary.js. This is where it executes Binary Actions. 1. It takes the committed polynomials A, B and C, and breaks them into bytes (in little-endian form). 2. It sequentially pushes each triplet of bytes (freeInA, freeInB, freeInC) into their corresponding registries (ai, bi, ci). It runs one for-loop for all committed polynomials (A, B, C), over all the bytes of the 8 registries, which are altogether 32 bytes per committed polynomial. Recall that LATCHSIZE = REGISTERSNUM BYTESPERREGISTER = 8 registries 4 bytes. It hence amounts to 32 bytes for each committed polynomial. 3. Once the 256-bit LATCH is built, it checks the opcodes and then computes the required binary operations in accordance with the instructions of the Main SM. 4. It also generates the final registries. The Binary PIL (program) There are two types of inputs to the Binary PIL program: the constant polynomials and the committed polynomials. The program operates byte-wise to carry out 256-bit Plookup operations. Each row of the lookup table is a vector of the form; \{ PLAST, POPCODE, PA, PB, PCIN, PC, PCOUT \}, consisting of the constant polynomials created by the Binary SM executor. As seen above, - PLAST is the Last-byte, - POPCODE is the 4-bit operation code, - PA and PB, are the 1-byte input polynomials, - PCIN is the 1-bit Carry-in, - PC is the 16-bit output polynomial, - PCOUT is the 1-bit Carry-out. The Binary PIL program takes in byte-size inputs, as in the Binary SM executor, each 256-bit input committed polynomial is first broken into 32 bytes. For each of the 32 triplets freeInA, freeInB and freeInC, tallying with the three 256-bit committed polynomials A,B and C, the Binary PIL program, 1. Prepares a Plookup input vector of the form; \{last, opcode, freeInA, freeInB, cIn, freeInC, cOut\}, where each element is a byte. 2. Runs Plookup, {last,opcode,freeInA,freeInB,cIn,freeInC,cOut} in {PLAST,POPCODE,PA,PB,PCIN,PC,PCOUT}; 3. Resets registry values at the end of the 32 cycles using RESET, and utilising FACTOR for correct placement of values. For e.g., $\texttt{a0' = a0 (1 - RESET) + freeInA FACTOR[0]}$; Special variables, useCarry and c0Temp, are used for managing updates and assignments of values, particularly for Boolean operations, where the output c0 registry value is either TRUE = 1 or FALSE = 0. Hence the Lines 104 and 105 of code; Line 104. c0Temp' = c0Temp (1 - RESET) + freeInC FACTOR[0]; Line 105. c0' = useCarry (cOut - c0Temp) + c0Temp; For all non-Boolean operations; the default value for useCarry is zero, making c0' = c0Temp. The value of c0' is therefore of the same form as other ci' update values. 4. The output of the Binary PIL program is therefore a report of either pass or fail. Source code The Polygon zkEVM repository is available on GitHub. Binary SM executor: smbinary.js Binary SM PIL: binary.pil Test vectors: binarytest.js # index.md: The design paradigm at Polygon has shifted to developing a zero-knowledge virtual machine that emulates the Ethereum Virtual Machine (EVM). Proving and verification of transactions in Polygon zkEVM are all handled by a zero-knowledge prover component called the zkProver. All the rules for a transaction to be valid are implemented and enforced in the zkProver. The zkProver performs complex mathematical computations in the form of polynomials and assembly language which are later verified on a smart contract. Those rules could be seen as constraints that a transaction must follow in order to be able to modify the state tree or the exit tree. !!!info zkProver is a component of the Polygon zkEVM which is solely responsible for Proving. Interaction with node and database The zkProver mainly interacts with two components, i.e. the Node and the database (DB). Hence, before diving deeper into other components, we must understand the flow of control between zkProver, the Node, and database. Here is a diagram to explain the process clearly. !zkProver, the Node, and Database As depicted in the flow diagram above, the whole interaction works out in 4 steps. 1. The node sends the content of Merkle trees to the database to be stored there. 2. The node then sends the input transactions to the zkProver. 3. The zkProver accesses the database and fetches the info needed to produce verifiable proofs of the transactions sent by the Node. This information consists of the Merkle roots, the keys and hashes of relevant siblings, and more. 4. The zkProver then generates the proofs of transactions, and sends these proofs back to the Node. However, this is really the tip of the iceberg in terms of what the zkProver does. There is a lot more detail involved in how the zkProver actually creates these verifiable proofs of transactions. It will be revealed while we dig deeper into state machines below. State machines The zkProver follows modularity of design to the extent that, except for a few components, it is mainly a cluster of state machines. It has a total of thirteen (13) state machines; 1. The Main state machine. 2. Secondary state machines; Binary SM, Storage SM, Memory SM, Arithmetic SM, Keccak Function SM, PoseidonG SM. 3. Auxiliary state machines; Padding-PG SM, Padding-KK SM, Bits2Field SM, Memory Align SM, Byte4 SM, ROM SM. Due to the modular design of zkProver, the Main state machine can delegate as many of tasks as possible to other specialist state machines. This heavily improves the efficiency of Main SM. Secondary state machines The Main SM executor directly instructs each of the secondary state machines by sending appropriate instructions called Actions, depicted in the below diagram. The grey boxes are not state machines but indicate Actions, which are specific instructions from the Main state machine to the relevant Secondary state machine. These instructions dictate how a state should transition in a state machine. However, every Action, whether from the generic Main SM or the specific SM, must be supported with a proof that it was correctly executed. !The Main SM Executor's Instructions There are some natural dependencies between; 1. The Storage state machine which uses merkle Trees and the Poseidon state machine, which is needed for computing hash values of all nodes in the Storage's Merkle Trees. 2. Each of the hashing state machines, Keccak Function SM and the PoseidonG SM, and their respective padding state machines, i.e. the Padding-KK SM and the Padding-PG SM. Two novel languages for zkProver The zkProver is the most complex module of zkEVM. It required development of two new programming languages to implement the needed elements; The Zero-Knowledge Assembly language and the Polynomial Identity Language. It is not surprising that the zkProver uses a language specifically created for the firmware and another for the hardware because adopting the state machines paradigm requires moving from high-level programming to low-level programming. These two languages, zkASM and PIL, were designed mindful of prospects for broader adoption outside Polygon zkEVM. Zero-knowledge assembly As an Assembly language, the Zero-Knowledge Assembly (or zkASM) language is specially designed to map instructions from the zkProver's Main state machine to other state machines. In case of the state machines with firmware, zkASM is the Interpreter for the firmware. Prescriptive assembly codes are generated by zkASM codes using instructions from the Main state machine to specify how a given SM Executor must carry out calculations. The Executor's strict adherence to the zkASM codes' logic and conventions makes computation verification simple. Polynomial identity language The Polynomial identity language (or PIL) is especially designed for the zkProver. Almost all state machines express computations in terms of polynomials. Therefore, state transitions in state machines must satisfy computation-specific polynomial identities. Polygon zkEVM is creating the most effective solution to solve the blockchain trilemma of privacy, security, and scalability. And its context is an efficient zero-knowledge commitment scheme. The most reliable and effective commitment schemes till-date are the Polynomial Commitment Schemes. Hence, it was convenient to transform calculations into some polynomial language, where verification essentially comes down to verifying whether execution fulfils specific polynomial identities. All PIL codes, in the zkProver's state machines, therefore form the very DNA of the verifier code. Microprocessor state machines There are two microprocessor-type state machines; the Main SM and the Storage SM. These two state machines have the Firmware and the Hardware part. It is worth noting that each of these Microprocessor SM has its own ROM. The Firmware part runs the zkASM language to set up the logic and rules, which are expressed in JSON format and stored in a ROM. The JSON-file is then parsed to the specific SM Executor, which executes Storage Actions in compliance with the rules and logic in the JSON-file. The Hardware component, which uses the Polynomial Identity Language (PIL), defines constraints (or polynomial identities), expresses them in JSON format, and stores them in the accompanying JSON-file. These constraints are parsed to the specific SM Executor, because all computations must be executed in conformance to the polynomial identities. !Microprocessor state machine !!!info Although the Main SM and the Storage SM have the same look and feel, they differ considerably. For example, the Storage SM specialises with execution of Storage Actions (also called SMT Actions), whilst the Main SM is responsible for a wider range of Actions. Nevertheless, the Main SM delegates most of these Actions to specialist state machines. Also, the Storage SM remains secondary as it receives instructions from the Main SM, and not conversely. Hashing in the zkProver There are two secondary state machines specialising with Hashing ; The Keccak state machine, and the Poseidon state machine. Each of them is an "automised" version of its standard cryptographic hash function. Keccak state machine The Keccak SM is a gates state machine. It has a set of logic gates (the hardware) and a set of connections between the gates (the logic). It's a secondary state machine composed of the Keccak SM Hash Generator and the Keccak PIL code, where the latter is for validation purposes. Poseidon state machine The POSEIDON hash function has been publicised as a zk-STARK-friendly hash function. It is newer than the Keccak hash and thus still under the scrutiny of cryptanalysts. As such, it is best-suited for the zkProver context. The Poseidon SM is very straight-forward if someone is familiar with the internal mechanism of the original Poseidon hash function. The hash function's permutation process translates readily to the state transitions of the POSEIDON state machine. The hash function's twelve (12) input elements, the non-linear substitution layers (the S-boxes) and the linear diffusion layers (the MDS matrices), are directly implemented in the state machine. Although a secondary state machine, the POSEIDON SM receives instructions from both the Main SM and the Storage SM. It has both the executor part and an internal PIL code (a set of verification rules), written in the PIL language. Proving execution-correctness The zkProver's state machines are designed to execute programs, as well as to guarantee that these programs are correctly executed. Each secondary state machine therefore consists of its own executor and a PIL program that can be used to check correct execution of all the instructions coming from the Main SM Executor. Here is a step-by-step outline of how the system achieves proof / verification of transactions: - Represent a given computation as a state machine (SM). - Express the state changes of the SM as polynomials. - Capture traces of state changes, called execution traces, as rows of a lookup table. - Form polynomial identities and/or constraints that these state transitions satisfy. - Prover uses a specific polynomial commitment scheme to commit and prove knowledge of the committed polynomials. - Plookup is one of the ways to check if the Prover's committed polynomials produce correct traces. While the polynomial constraints are written in the PIL language, the instructions are initially written in zkASM but subsequently expressed and stored in JSON format. Although not all verification involves a Plookup, the diagram below, briefly illustrates the wide role Plookup plays in the zkProver. !Plookup and the zkProver state machines Components of zkProver For the sake of simplicity, one can think of the zkProver as being composed of the following four components; - The Executor or the Main state machine executor. - The STARK Recursion Component. - The CIRCOM library. - The zk-SNARK Prover. In the nutshell, the zkProver uses these four components to generates verifiable proofs. As a result, the constraints that each proposed batch must meet are polynomial constraints or polynomial identities. All valid batches must satisfy specific polynomial constraints. !Simplified Diagram of zkProver The executor The Executor or Main state machine Executor handles the execution of the zkEVM. This is where EVM Bytecodes are interpreted using a new zero-knowledge Assembly language (or zkASM), specially developed by the Polygon zkEVM team. It takes as inputs; the transactions, the old and the new states, the ChainID of the Sequencer, to mention a few. The executor also needs; 1. The PIL, which is the list of polynomials, the list of the registers. 2. The ROM, which stores the list of instructions pertaining to execution. The Executor sets up the polynomial constraints that every valid batch of transactions must satisfy. Another language, specially developed by the team, called Polynomial Identity Language (or PIL), is used to encode all the polynomial constraints. The Executor executes all instructions on top of the PIL hardware and generates the committed polynomials; which are the state machine cycles, or a list of all the states. It also generates some public data, which forms part of the input to the zk-SNARK verifier. STARK recursion component Once the Main state machine Executor has converted transactions and related data to committed polynomials, the STARK Recursion component takes the following inputs; 1. The committed polynomials. 2. The constant polynomials. 3. Scripts, which are lists of instructions. These are taken in order to generate a zk-STARK proof. In an effort to facilitate fast zk-STARK proving, the STARK Recursion Component utilises Fast Reed-Solomon Interactive Oracle Proofs of Proximity (RS-IOPP), also referred to as FRI, for each zk-proof. The component is referred to as the STARK Recursion, because; - It actually produces several zk-STARK proofs. - Collates them into bundles of a few zk-STARK proofs. - Produces a further zk-STARK proof for each bundle. - The resulting zk-STARK proofs of the bundles are also collated and proved with only one zk-STARK proof. This way, hundreds of zk-STARK proofs are represented and proved with only one zk-STARK proof. CIRCOM library The single zk-STARK proof produced by the STARK Recursion Component is the input to a CIRCOM component. The original CIRCOM paper describes it as both a circuits programming language to define Arithmetic circuits, and a compiler that generates two things; 1. A file containing a set of Rank-1 Constraints System (or R1CS) constraints associated with an Arithmetic circuit. 2. A program (written either in C++ or WebAssembly) for computing a valid assignment to all wires of the Arithmetic circuit, called a witness. As implemented in the zkProver, CIRCOM takes as input a zk-STARK proof in order to perform two tasks; - Defining and creating an Arithmetic circuit corresponding to the input zk-STARK proof. This Arithmetic circuit is expressed in terms of its equivalent R1CS. - Generating the witness, which is in fact a set of valid input, intermediate and output values of the circuit wires satisfying the above R1CS. zk-SNARK prover The last component of the zkProver is the zk-SNARK Prover, in particular, Rapid SNARK. Rapid SNARK is a zk-SNARK proof generator, written in C++ and Intel Assembly, which is very fast in generating proofs of CIRCOM's outputs. With regards to the zkProver, the Rapid SNARK takes as inputs 1. The witness from CIRCOM. 2. The STARK verifier data, which dictates how the Rapid SNARK must process the data, and then generate a zk-SNARK proof. Strategy to achieving succinctness zk-STARK proofs are used because of their speed, and they require no trusted setup. They are however much larger compared to zk-SNARK proofs. It is for this reason, and the succinctness of the zk-SNARKs, that the zkProver uses a zk-SNARK to attest to the correctness of the zk-STARK proofs. zk-SNARKs are therefore published as the validity proofs to state changes. This strategy has huge benefits as it results in gas costs reducing from 5M to 350K. # mem-align-sm.md: The Memory-align state machine is a secondary state machine that includes an executor (the Memory-align SM executor) and an internal Memory-align PIL (program) that is a set of verification rules written in the PIL language. The Memory-align SM executor is written in two languages: Javascript and C/C++. Overview It checks memory reads/writes using a 32-byte word access, while the EVM can read and write 32-byte words with offsets at a byte level. The table below shows a sample of possible byte-addressed and 32-byte-addressed memory layouts for the same content (three words). $$ \begin{array}{|c|c|} \hline \mathbf{ADDRESS} &\mathbf{BYTE} \\ \hline \mathtt{0} &\mathtt{0xc4} \\ \mathtt{1} &\mathtt{0x17} \\ \mathtt{\vdots} &\mathtt{\vdots} \\ \mathtt{30} &\mathtt{0x81} \\ \mathtt{31} &\mathtt{0xa7} \\ \mathtt{32} &\mathtt{0x88} \\ \mathtt{33} &\mathtt{0xd1} \\ \mathtt{\vdots} &\mathtt{\vdots} \\ \mathtt{62} &\mathtt{0xb7} \\ \mathtt{63} &\mathtt{0x23} \\ \hline \end{array} $$ $$ \begin{array}{|c|c|} \hline \textbf{ADDRESS} & \textbf{32-BYTE WORD} \\ \hline \mathtt{0} &\mathtt{0xc417...81a7} \\ \mathtt{1} &\mathtt{0x88d1...b723} \\ \hline \end{array} $$ The relationship between the 32-byte word addressable layout and the byte addressable layout is called memory alignment, and the Memory-align SM is the state machine that checks the correctness of this relationship. In more detail, we have to check the following memory operations: - $\mathtt{MLOAD}$: It receives an offset and returns the 32 bytes in memory starting at that offset. - $\mathtt{MSTORE}$: It receives an offset and saves 32 bytes from the offset address of the memory. - $\mathtt{MSTORE8}$: It receives an offset and saves one byte on that address of the memory. In general cases, $\mathtt{MLOAD}$ requires reading bytes of two different words. Given that the content of memory is represented as shown in the above table, and that the EVM is addressed at the byte level, the value we should get when checking a read from the EVM of a word starting at the address $\mathtt{0x22}$ is as follows: $$ \mathtt{val} = \mathtt{0x1f \cdots b7236e21}. $$ We denote the content of the words affected by an EVM memory read as $\mathtt{m}0$ and $\mathtt{m}1$. In our example, these words are the following: $$ \mathtt{m}0 = \mathtt{0x} \mathtt{88d11f} \cdots \mathtt{b723}, \quad \mathtt{m}1 = \mathtt{0x} \mathtt{6e21ff} \cdots \mathtt{54f9}. $$ We define a read block as the string concatenating the content of the words affected by the read: $\mathtt{m}0 \mid \mathtt{m}1$. The below figure shows the affected read words $\mathtt{m}0$ and $\mathtt{m}1$ that form the affected read block and the read value $\mathtt{val}$ for a read from the EVM at address $\mathtt{0x22}$ in our example memory. !Schema of MLOAD Example Let us now introduce the flow at the time of validating a read. Suppose that we want to validate that if we perform an $\mathtt{MLOAD}$ operation at the address $\mathtt{0x22}$, we get the previous value $\mathtt{0x1f\dotsb7236e21}$. At this point, the Main state machine performs several operations. First of all, it has to query for the values $\mathtt{m}0$ and $\mathtt{m}1$. Henceforth, it must call the Memory SM in order to validate the previous queries. Observe that it is easy to extract the memory positions to query from the address $\mathtt{0x22}$. In fact, if a is the memory position of the $\mathtt{MLOAD}$ operation, then $\mathtt{m}0$ is always stored at the memory position $\lfloor \frac{a}{32} \rfloor$ and $\mathtt{m}1$ is stored at the memory position $\lfloor \frac{a}{32} \rfloor + 1$. In our example, $a = \mathtt{0x22} = 34$. Hence, $\mathtt{m}0$ is stored at the position $\lfloor \frac{32}{34} \rfloor = \mathtt{0x01}$ and $\mathtt{m}1$ is stored at the position $\lfloor \frac{32}{34} \rfloor + 1= \mathtt{0x02}$. Secondly, we should extract the correct $\mathtt{offset}$. The $\mathtt{offset}$ represents an index between 0 and 31 indicating the number of bytes we should offset from the starting of $\mathtt{m}0$ to correctly place $\mathtt{val}$ in the block. In our case, the $\mathtt{offset}$ is 2. Similarly as before, it is easy to obtain the offset from a. In fact, the it is equal to a $(\mathrm{mod} \ 32)$. Now, the Main SM checks, via a Plookup to the Memory-align state machine, that $\mathtt{val}$ is a correct read given the affected words $\mathtt{m}0$ and $\mathtt{m}1$ and the $\mathtt{offset}$.

That is, we should check that the value $\mathtt{val}$ can be correctly split into $\mathtt{m}0$ and $\mathtt{m}1$ using the provided $\mathtt{offset}$. Similarly, $\mathtt{MSTORE}$ instruction requires writing bytes in two words. The idea is very similar, but we are provided with a value \val that we want to write into a specific location of the memory. We denote by $\mathtt{w}0$ and $\mathtt{w}1$ the words that arise from $\mathtt{m}0$ and $\mathtt{m}1$ after the corresponding write. Following our previous example, suppose that we want to write: $$ \mathtt{val} = \mathtt{0xe201e6\dots662b} $$ in the address $\mathtt{0x22}$ of the byte-addressed Ethereum memory. We are using the same $\mathtt{m}0$ and $\mathtt{m}1$ (and since we are writing into the same address as before) and they transition into: $$ \mathtt{w}0 = \mathtt{0x88d1} \mathtt{\e201e6\dots},\quad \mathtt{w}1 = \mathtt{0x} \mathtt{662b} \mathtt{ff\dots54f9}. $$!Schema of MSTORE example The Main state machine needs to perform several operations. Given an address $\mathtt{addr}$, an offset value $\mathtt{offset}$ and a value to be written $\mathtt{val}$. As seen before, the Main SM is in charge of reading the zkEVM memory to find $\mathtt{m}0$ and $\mathtt{m}1$ from the given address and offset. Of course, the validity of this query should be performed with a specific Plookup into the Memory SM, as before. Now, the Main SM can compute $\mathtt{w}0$ and $\mathtt{w}1$ from all the previous values in a unique way. The way of validating that we are providing the correct $\mathtt{w}0$ and $\mathtt{w}1$ is to perform a Plookup into the Memory-align SM. That is, we check that the provided values $\mathtt{w}0$ and $\mathtt{w}1$ are correctly constructed from the provided $\mathtt{val}$, $\mathtt{m}0$, $\mathtt{m}1$ and $\mathtt{offset}$ values. Finally, the last opcode $\mathtt{MSTOREE}$ works similarly, but it only affects one word $\mathtt{m}0$. Moreover, we can only write one byte and hence, only the less significant byte of $\mathtt{val}$ is considered into the write. Observe that, in this opcode, $\mathtt{m}1$ and $\mathtt{w}1$ are unconstrained. Source code The Polygon zkEVM repository is available on GitHub. Memory-align SM executor: sm_mem_align.js Memory-align SM PIL: mem_align.pil # memory-sm.md: The Memory State Machine is a secondary state machine, therefore it has the executor part (the Memory SM executor) and an internal Memory PIL program which is a set of verification rules, written in the PIL language. The Memory SM executor is written in two languages: Javascript and C/C++. EVM memory The memory of the Ethereum Virtual Machine (EVM) is a volatile read-write memory, and it is therefore used to store temporary data during the execution of transactions when smart contracts are deployed. That is, data in memory is populated during transaction execution but it does not persist between transactions. The memory is an array of 256-bit (32 bytes) words that can be accessed through addresses at byte level, that is to say, each byte in the memory has a different address. Memory has addresses of 32 bits, and initially, all memory locations are composed by bytes set to zero. Now, let's see the layout in memory of the following two words $\texttt{0xc417...81a7}$ and $\texttt{0x88d1...b723}$. The below table displays this layout. Let's call this Table 1.

| $\mathbf{ADDRESS}$ | $\mathbf{BYTE}$ | | :----------------: | :---------------: | | $\mathtt{0}$ | $\mathtt{0xc4}$ | | $\mathtt{1}$ | $\mathtt{0x17}$ | | $\mathtt{\vdots}$ | $\mathtt{\vdots}$ | | $\mathtt{30}$ | $\mathtt{0x81}$ | | $\mathtt{31}$ | $\mathtt{0xa7}$ | | $\mathtt{32}$ | $\mathtt{0x88}$ | | $\mathtt{33}$ | $\mathtt{0xd1}$ | | $\mathtt{\vdots}$ | $\mathtt{\vdots}$ | | $\mathtt{62}$ | $\mathtt{0xb7}$ | | $\mathtt{63}$ | $\mathtt{0x23}$ |

Observe that each word has 32 bytes and the words are stored in Big-Endian form. So, the most significant bytes are set in the lower addresses. The EVM provides three opcodes to interact with the memory area. There is an opcode to read, and an opcode to write 32-byte words providing an offset: - $\texttt{MLOAD}$: It receives an offset and returns the 32 bytes in memory starting at that offset - $\texttt{MSTORE}$: It receives an offset and saves 32 bytes from the offset address of the memory Considering our previous memory contents, if we perform an $\texttt{MLOAD}$ with an offset of $\texttt{1}$, we would obtain the following word: $\texttt{0x17...a788}$. On the other hand, if we do an $\texttt{MSTORE}$ with an offset of $\texttt{1}$ with the word $\texttt{0x74f0...ce92}$, we would modify the content of the memory as shown in the below table (or Table 2).

| $\mathbf{ADDRESS}$ | $\mathbf{BYTE}$ | | :----------------: | :----------------------: | | $\mathtt{0}$ | $\mathtt{0xc4}$ | | $\mathtt{1}$ | $\mathtt{\textbf{0x74}}$ | | $\mathtt{2}$ | $\mathtt{\textbf{0xf0}}$ | | $\mathtt{\vdots}$ | $\mathtt{\vdots}$ | | $\mathtt{30}$ | $\mathtt{\textbf{0xce}}$ | | $\mathtt{31}$ | $\mathtt{\textbf{0x92}}$ | | $\mathtt{33}$ | $\mathtt{0xd1}$ | | $\mathtt{\vdots}$ | $\mathtt{\vdots}$ | | $\mathtt{62}$ | $\mathtt{0xb7}$ | | $\mathtt{63}$ | $\mathtt{0x23}$ |

When the offset is not a multiple of 32 (or 0x20), as in the previous example, we have to use bytes from two different words when doing $\texttt{MLOAD}$ or $\texttt{MSTORE}$. Finally, the EVM provides a write memory operation that just writes a byte: - $\texttt{MSTOREE}$: It receives an offset and saves one byte on that address of the memory. $\texttt{MSTOREE}$ always uses only one word. Layout The Memory SM is in charge of proving the memory operations in the execution trace. As mentioned, read and write operations use addresses at byte level in the EVM. However, doing the proofs byte-by-byte would consume many values in the trace of this state machine. Instead, in this machine, we operate addressing words (32 bytes). For example, if we have the memory layout from Table 1, then we would have the memory layout of the below provided table with addresses that point to 32-byte words.

| $\textbf{ADDRESS}$ | $\textbf{32-BYTE WORD}$ | | :----------------: | :---------------------: | | $\mathtt{0}$ | $\mathtt{0xc417...81a7}$ | | $\mathtt{1}$ | $\mathtt{0x88d1...b723}$ |

The Memory SM uses this latter layout, the 32-byte word access, to check reads and writes. However, as previously mentioned, the EVM can read and write with offsets at a byte level. As a result, we will need to check the relationship between byte access and 32-byte word access. For these checks, we have another state machine called Memory-align SM. Design As with any state machine, the Memory SM has an executor to compute the trace that proves the correctness of memory reads and writes and a PIL description that enforces that the trace is correct. Execution trace design The Memory SM defines the design of the trace and the PIL description that checks that memory reads and writes aligned to 32-byte words are correct. The addresses, denoted as $\texttt{addr}$, are represented through 32 bits (4 bytes) and point to 32-byte words. The value of words stored in memory, denoted as $\texttt{val}$, are represented through 8 registers $\texttt{val[0..7]}$ of 4 bytes each, making a total of 32 bytes (256 bits). The below table shows an example with all the memory operations present at an execution trace of the Main SM.

| $\texttt{step}$ | $\texttt{mOp}$ | $\texttt{mWr}$ | $\texttt{addr}$ | $\texttt{val[7]}$ | $\texttt{val[6]}$ | $\dots$ | $\texttt{val[0]}$ | | :-------------: | :------------: | :------------: | :-------------: | :---------------: | :---------------: | :-----: | :---------------
| 11 | 1 | 1 | 6 | 2121 | 3782 | $\dots$ | 5432 | | 31 | 1 | 1 | 4 | 3231 | 9326 | $\dots$ | 8012 | | 55 | 1 | 0 | 6 | 2121 | 3782 | $\dots$ | 5432 | | 63 | 1 | 1 | 6 | 4874 | 1725 | $\dots$ | 2074 | | 72 | 1 | 0 | 4 | 3231 | 9326 | $\dots$ | 8012
89 | 1 | 1 | 2 | 9167 | 5291 | $\dots$ | 6001 |

The $\texttt{step}$ is the execution step number at the Main SM and in this case, we are showing only the steps that are performing a memory operation. The instruction to execute a memory operation is indicated by the $\texttt{mOp}$ selector. The $\texttt{mWr}$ is also a selector that shows whether the memory operation is a read or a write. In the previous trace, we can observe that the first memory operation is performed at step 11 and it is the write of the sixth 32-byte word. The eight registers $\texttt{val[0..7]}$ provide the bytes to be written in that word. It is worth to mention that for a specific word address, the first operation is always a write because it makes no sense to read a position that has not been previously written. Then, in this word address there can be a sequence of reads and writes. In the previous trace, we can observe that for the sixth word, there is a write at step 11, then a read at step 55 and finally another write at step 63.

| $\texttt{step}$ | $\texttt{addr}$ | $\texttt{mOp}$ | $\texttt{mWr}$ | $\texttt{val[7]}$ | $\texttt{val[6]}$ | $\dots$ | $\texttt{val[0]}$ | | :-------------: | :-------------: | :------------: | :------------: | :---------------: | :---------------: | :-----: | :---------------
| 89 | 2 | 1 | 1 | 9167 | 5291 | $\dots$ | 6001 | | 31 | 4 | 1 | 1 | 3231 | 9326 | $\dots$ | 8012 | | 72 | 4 | 1 | 0 | 3231 | 9326 | $\dots$ | 8012 | | 11 | 6 | 1 | 1 | 2121 | 3782 | $\dots$ | 5432 | | 55 | 6 | 1 | 0 | 2121 | 3782 | $\dots$ | 5432
63 | 6 | 1 | 1 | 4674 | 1725 | $\dots$ | 2074 |

The trace of the Memory SM must check that the writes are done according to their step and that reads provide the correct words according also to their step. In order to implement these checks, the execution trace of the Memory SM sorts all the memory operations; firstly by $\texttt{addr}$, and secondly by $\texttt{step}$, as shown in the above table. This ordering is referred to as the topology of the Memory SM. Finally, we will need to add a few more columns to ensure that the memory execution trace goes exactly across all the ordered writes and reads of the Main SM, with writes storing the provided values and with reads not changing the previous value of the word. In particular, we add three more columns. One of these columns is called $\texttt{INCS}$ and it is used to provide an order for the values of the columns. Another column called $\texttt{lastAccess}$ is used to enable an address change when all the memory operations for this address have appeared at the trace. The last column is called $\texttt{ISNOTLAST}$ and it is used to make the checks pass when there are no more memory accesses. List of columns The following columns (polynomials) are used by the Memory SM. We divide them between preprocessed and committed polynomials. Preprocessed - $\texttt{INCS}$: Counter that goes from 1 up to N, where N is the number of rows in the computational trace, $$ \texttt{INCS} = (\underbrace{1, 2, 3, \dots,N-1, N}{N}) $$ It is used to do a range check to prove the incremental order of other columns. - $\texttt{ISNOTLAST}$: Selector that is 1 in every row except in the N-th row, in which its value is 0, $$ \texttt{ISNOTLAST} = (\underbrace{1, 1, 1, \dots,1, 0}{N}) $$ Committed - $\texttt{step}$: Position in which the memory operation was called in the Main SM. - $\texttt{mOp}$: Selector indicating whether it is being performed a memory operation or not. - $\texttt{mWr}$: Selector that is 1 if the memory operation is a write and 0 if it is a read operation. - $\texttt{addr}$: A 4-byte (or 32 bit) integer indicating the address of a 32-byte word. - $\texttt{lastAccess}$: Selector indicating whether it has been reached the last memory access for a particular address or not. - $\texttt{val[0..7]}$: Vector containing 8 4-byte integer indicating the 256-bit value associated to a given address. Complete example The table below shows the complete Memory SM trace for our example in which the computational trace size N is 2^3. There are various important details to remark from the point in which all memory accesses have been completed but the 2^3-th row has not been reached yet: - $\texttt{mOp}$ and $\texttt{mWr}$ are set to 0 until the last row. - $\texttt{addr}$ is incremented by 1 to keep the incremental order of the addresses. This value is maintained until the last row. - $\texttt{lastAccess}$ is also set to 0 except in the very last row, where it is set back to 1 to create the ciclycity behavior. - $\texttt{step}$ is incremented by 1 in each row so that this column fulfills the constraints describing this state machine. - $\texttt{Remark}$. Notice that $\texttt{step}$ can take values beyond N and that the value of $\texttt{step}$ after the row of the last address can coincide with a previous value. As we will show in the next section, where we describe the constraints, these facts do not cause any issue. - $\texttt{val[0..7]}$ are all set to 0 until the last row. !Complete Memory SM Execution Trace Constraints What constraints does the execution trace have to satisfy? And how is the Memory SM connected with the Main State Machine. Topology Let's start with the set of constraints regarding the topology of the state machine. !eqn set 1 Equations (1) and (2) are straightforward. Equation (1) asserts that $\texttt{lastAccess}$ is a selector (i.e., a column whose values lie in the set $\{0,1\}$), while Equation (2) confirms that $\texttt{addr}$ does not change until it is accessed for the last time. Note that Equation (2) implies that addresses are processed one-by-one in the Memory SM, but it does not guarantee that they are ordered incrementally. Equation (3) is a little bit more tricky. Let's do a case analysis on it. The curly braces notation in Equation (3) means that the inclusion is only checked at values such that the corresponding selector $\texttt{ISNOTLAST}$ is equal to 1. Then, depending on the value of the $\texttt{lastAccess}$ selector we have two cases: - If $\texttt{lastAccess} = 0$, $$ \texttt{step}' - \texttt{step} \subset \texttt{INCS}. $$ - Else, $$ \texttt{addr}' - \texttt{addr} \subset \texttt{INCS}. $$ In words, whenever a transition do not change the address in question, verify that $\texttt{step}' > \texttt{step}$; otherwise verify that $\texttt{addr}' > \texttt{addr}$. Therefore, Equation (3) ensures that both $\texttt{step}$ and $\texttt{addr}$ are ordered incrementally (first by $\texttt{addr}$ and then by $\texttt{step}$). A combination of Eqs. (2) and (3) gives the desired topology. Operation selectors Let's continue with the operation selectors: $\texttt{mOp}$ and $\texttt{mWr}$. $$ \texttt{mOp} \cdot (\texttt{mOp} - 1) = 0 \ \ \tag{4} $$ $$ \texttt{mWr} \cdot (\texttt{mWr} - 1) = 0 \ \ \tag{5} $$ $$ (1 - \texttt{mOp}) \cdot \texttt{mWr} = 0 \ \ \tag{6} $$ Eqs. (4) and (5) ensure that $\texttt{mOp}$ and $\texttt{mWr}$ are, effectively, selectors. Eq. (6) is imposing a restriction to $\texttt{mWr}$ (and binding it with $\texttt{mOp}$) in the following sense: - $\texttt{mWr}$ can be set to 1 only if $\texttt{mOp}$ is also set to 1. - Similarly, if $\texttt{mOp}$ is set to 0, then $\texttt{mWr}$ should be set to 0 as well. This restriction comes naturally from the definition of these selectors. Updating the value Finally, we explain the constraints that deal with the value columns $\texttt{val[0..7]}$. !eqn set 3 We analyze both Eqs. (7) and (8) at the same time. Notice that we simply discuss the feasible cases: - Maintain the same value when reading: If $\texttt{mWr}' = 0$ and $\texttt{lastAccess} = 0$, then it should be the case that $\texttt{val[0..7]}' = \texttt{val[0..7]}$, since this means that we will perform a read in the next step. Eq. (7) ensures this case. - Filling the value with zeros when done: If $\texttt{mOp}' = 1$, $\texttt{mWr}' = 0$ and $\texttt{lastAccess} = 1$, then it should be the case that $\texttt{val[0..7]}' = 0$, i.e., the register $\texttt{val[0..7]}$ is set to 0 in the forthcoming steps. More cases are not possible because for an address we always start with a write operation which limits the behavior of these constraints to these cases. For example, it's impossible for $\texttt{lastAccess = 1}$ when $\texttt{mOp}'$ or $\texttt{mWr}'$ is 0; because the first operation that is always performed over a memory address is a write. However, notice that to be able reset $\texttt{addr}$ to its state in the first row (where it is the case that $\texttt{addr}' < \texttt{addr}$) it should be the case that $\texttt{lastAccess} = 1$ in the last row of the computational case. If $\texttt{lastAccess}$ would have not been set to 1, then Eq. (2) would not be satisfied. We obtain this condition by adding the following constraint: $$ \left(1 - \texttt{lastAccess}\right) \cdot \left(1 - \texttt{ISNOTLAST}\right) = 0. $$ Connection with the Main SM The last constraint that we need to add is to relate the execution trace of the Main SM with the execution trace of the Memory SM. The constraint has to check that all the rows in the trace of the Main SM that make memory operations (i.e. rows where $\texttt{mOp} == 1$) are a permutation (any permutation) of the rows of the Memory SM where $\texttt{mOp} == 1$. The key point is that if both vectors would not be a permutation of each other, then that would mean that the Main SM is performing an incorrect memory action. Source code The Polygon zkEVM repository is available on GitHub. Memory SM executor: sm_mem.js Memory SM PIL: mem.pil # bits2field-sm.md: The Bits2Field state machine is one of the auxiliary state machines used specifically for parallelizing the implementation of KECCAK-F SM. Its source code is available here. The Bits2Field state machine ensures correct packing of $\mathtt{44}$ bits from $\mathtt{44}$ different $\mathtt{1600}$-row blocks of the Padding-KK-Bit SM into a single field element. Therefore, it operates like a (44-bits-to-1-field-element) multiplexer between the Padding-KK-Bit SM and the Keccak-F SM. In simpler terms, it takes bits from $\mathtt{44}$ different blocks, places them into the first 44 bit-positions of a single field element, whereupon the KECCAK-F circuit runs. The name Bits2Field state machine refers to the process of taking 44 bits from 44 different blocks of the Padding-KK-Bit SM and inserting them into a single field element. Although the KECCAK-F SM is a binary circuit, instead of executing on a bit-by-bit basis, it is implemented to execute KECCAK-F operations on a 44bits-by-44bits basis. This is tantamount to running $\mathtt{44}$ KECCAK-F hashing circuits in parallel. The figure below depicts a simplified multiplexing process. !The 44 bits to 1 field-element Multiplexing Mapping 44 Bits To A 64-bit Field Element Suppose operations are carried out in a field $\mathbb{F}p$ of $\mathtt{64}$-bit numbers. The smallest field used in the zkProver is the Goldilocks Field $\mathbb{F}p$ where $p = 2^{64} - 2^{32}+1$. After multiplexing, the 44 bits are loaded into the first 44 least significant bit-positions of the field element as depicted in the figure below. !Figure 2: 44 Bits mapped to a 64-bit field element A field element as an input to the KECCAK-F circuit is of the form, $$ \mathtt{0b}\mathtt{0000\ 0000\ 0000\ 0000\ 0000}\ \mathtt{X}1 \mathtt{X}2 \mathtt{X}3 \mathtt{X}4\ \mathtt{X}5 \mathtt{X}6 \mathtt{X}7 \mathtt{X}8\ \dots \mathtt{X}{43} \mathtt{X}{44} \text{ } $$ and it is composed of 20 zeroes and 44 meaningful bits related to the committed polynomials. Given the capacity of 2^{23} in terms of the state machine evaluations (i.e., the degree of polynomials) and the KECCAK-F's $\texttt{SlotSize} = 155286$, one obtains $2^{23} / 155286 = 54.020375307$ KECCAK-F slots. Therefore, a total of 54 slots $\times$ 44 blocks $= 2376$ Keccak blocks can be processed. This is a big improvement from the previous 477 blocks of the 9bits-to-1field element multiplexing (i.e., $53 \times 9 = 477$). The Bits2Field PIL Code The Bits2Field executor executes the multiplexing of forty-four $\mathtt{1600}$-bit blocks into $\mathtt{1600}$ field elements, where each is a $\mathtt{N}$-bit field element. For reference, see the above figure where $\mathtt{N = 64}$. The question here is how to identify each of the original 9 bits of the field element to track their corresponding resultant $\mathtt{XOR}$ values or $\mathtt{ANDP}$ values? Note that every bit $\mathtt{b{i,j}}$ from the $\mathtt{i}$-th $\mathtt{1600}$-bit block is placed at the $\mathtt{2^{i}}$-th position of the $\mathtt{N}$-bit field element. The PIL code therefore uses factors denoted by $\mathtt{Factor}$, such that $\mathtt{Factor \in \{ 1, 2, 4, \dots , 2^{43} \}}$, and a $\mathtt{Fieldlatch}$ after running through forty-four $\mathtt{1600}$-bit blocks. Suppose $\mathtt{N = 64}$. Then the 44 least significant bit-positions of the field element looks like this: $$ \mathtt{field44 = X1 \cdot 2^{43} + X{2}{2}^{42} + X{3}{2}^{41} + \dots + X{42}{2}^2 + X{43}{2} + X{44}}. $$ The constraint checked is therefore, $$ \mathtt{field44' = (1-Fieldlatch)field44 + bitFactor} $$ The accumulated field element at the end of the execution (every forty-fourth row of the execution trace) is checked against the KECCAK-F input $\mathtt{KeccakF.a}$ with the boundary constraint, $$ \mathtt{Fieldlatch(field44 - KeccakF.a) = 0} $$ The PIL code is given below. % "bits2field.pil" include "keccakf.pil"; namespace Bits2Field(%N); pol constant FieldLatch; // [0:44,1] pol constant Factor; // 1,2,4,8,...,2^43 pol commit bit; pol commit field44; field44' = (1-FieldLatch)field44 + bitFactor; bit (1-bit) = 0; FieldLatch(field44 - KeccakF.a44) = 0; # index.md: For hashing, the zkEVM utilizes two state machines: the Keccak state machine and the Poseidon state machine. The Keccak-256 hash function is used for seamless EVM compatibility, whereas Poseidon is best suited for the zkProver context because it is a STARK-friendly hash (SFH) function. The sponge construction By design, Keccak and Poseidon are both sponge constructions. A generic sponge construction is a simple iterated construction for building a function: $$ F: \mathbb{Z}^ \to \mathbb{Z}^l $$ with an input of variable-length and arbitrary output length based on a fixed-length permutation: $$ f: \mathbb{Z}^b \to \mathbb{Z}^b $$ operating on a fixed number b of bits. The array of b bits that f keeps transforming is called the state, and b is called the width of the state. The state array is split into two chunks, one with r bits and the other with c bits. The width $b = r + c$, where r is called the bitrate (or simply rate) and c is called the capacity. Sponge construction phases The elements that completely describe a single instance of a sponge construction are: the fixed-length permutation f, the padding rule pad, the bitrate value r, and the capacity c. A schema of the sponge construction is shown in the below figure. !A Sponge Function Construction Initializing phase The input string is either padded to reach the r-bit length (if it was shorter than r bits) or split into r-bit long chunks, with the last one padded to reach the r-bit length (if it was longer than r bits). A hash function-specific reversible padding rule is used. The state of the hash function is initialized to a b-bit vector (or array) of zeros. Absorbing phase During this phase, the r-bit input blocks are XOR-ed sequentially with the first r bits of the state, intermixed with permutation function f applications. This process is repeated until all input blocks have been XOR-ed with the state. Take note that the last c bits, which correspond to the capacity value, do not absorb any external input. Squeezing phase The first r bits of the state are returned as output blocks in this phase, intermixed with applications of the function f. The number of output blocks is entirely up to the user. Keep in mind that the last c bits, which correspond to the capacity value, are never output during this phase. Actually, if the output is longer than the specified length, it is truncated to the required size. # keccak-framework.md: The zkEVM, as an L2 zk-Rollup for Ethereum, employs the Keccak hash function to achieve seamless compatibility with the Ethereum blockchain at Layer 1. However, rather than implementing a single state machine that performs four different tasks, the zkEVM does so in a framework of four state machines: 1. The Padding-KK SM is used for padding purposes, as well as validation of hash-related computations pertaining to the Main SM's queries. As depicted in the figure below, the Padding-KK SM is the Main SM's gateway to the Keccak hashing framework. !Keccak Design Schema 2. The Padding-KK-Bit SM converts between two string formats, the bytes of the Padding-KK SM to the bits of the Keccak-F Hashing SM, and vice-versa. 3. The Bits2Field SM is used specifically for parallelizing Keccak-F SM implementation. It acts as a multiplexer between the Padding-KK-Bit SM and the Keccak-F SM. This state machine is called Bits2Field because it ensures correct packing of bits from 44 different blocks of the Padding-KK-Bit SM into a single field element. In other words, it is a 44-bits-to-1-field-element multiplexer. 4. The Keccak-F SM computes string hashes at the request of the Main SM. Although the Keccak-F SM is a binary circuit, it operates on a 44-bits-by-44-bits basis rather than a bit-by-bit basis. This equates to running four 44 hashing circuits in parallel. Source Code - Padding-KK State Machine - Padding-KK-Bit SM - Bits2Field SM - Keccak-F SM # keccakf-sm.md: The Keccak-F state machine is one of the secondary zkProver state machines. It computes message string hashes and validates the accuracy of those computations upon the request of Main SM. Although the architecture of the original Keccak hash function is simple, the Keccak-F SM is not just a state machine or a simple automated version of the cryptographic hash function. First, a gates state machine is used to implement the Keccak-F SM. A binary circuit that operates on a bit-by-bit basis. Second, with regard to the Main SM, the Padding-KK-Bit SM, and the Bits2Field SM together provide a framework within which the Keccak-F SM is realized. Thirdly, the Keccak-F circuit is constructed in such a way that executing it once is effectively equivalent to operating forty-four (44) hashing circuits simultaneously, particularly in the first version of the zkEVM public testnet. For further information on how this parallelism technique is applied, see the Bits2Field SM. The Keccak-F circuit is briefly described in this article, along with a thorough explanation of the widely used Keccak-256 hash function and its specific parameters as they apply to the Polygon zkEVM implementation. Keccak-F circuit The Keccak-F circuit has two types of gates, types $\mathtt{0}$ and $\mathtt{1}$, corresponding to the two binary operations it performs, the $\mathtt{XOR}$ and $\mathtt{ANDP}$. The Keccak-F executor builds the constant polynomials, $\mathtt{ConnA}$, $\mathtt{ConnB}$ and $\mathtt{ConnC}$, and these are to be tested if they match their corresponding polynomials, $\mathtt{kA}$, $\mathtt{kB}$ and $\mathtt{kC}$. Also, for any $\textit{op} \in \{ \mathtt{XOR}, \mathtt{ANDP} \}$ we have, $$ \textit{op} \big(\mathtt{ConnA}, \mathtt{ConnB}\big) = \mathtt{ConnC} \\ \textit{op} \big(\mathtt{kA}, \mathtt{kB} \big) = \mathtt{kC} $$ The 44-bits are loaded into the state machine as 11-bit chunks. In keccakf.pil, each committed polynomial $\texttt{a[4]}$ is expressed in terms of 4 chunks, where each is 11 bits long. The corresponding a 44-bit array can be expressed as, $$ \texttt{a44} = \texttt{a}[3] \cdot 2^{33} + \texttt{a}[2] \cdot 2^{22} + \texttt{a}[1] \cdot 2^{11} + \texttt{a}[0] $$ where $\texttt{a}[i]$ for each $i \in \{ 0, 1, 2, 3 \}$. The verification involves a copy constraint $$ \mathtt{ \{a44, b44, c44\}}\ \texttt{connect}\ \mathtt{\{ConnA, ConnB, ConnC\}}; $$ and a Plookup as, $$ \texttt{ \{GateType, a[i], b[i], c[i]\}}\ \texttt{in} \ \mathtt{\{kGateType, kA, kB, kC\}}; $$ for each $i \in \{ 0, 1, 2, 3 \}$. This covers the Keccak-F circuit in a nutshell together with its PIL code. See the codes of smkeccakf.js and keccakf.pil on GitHub. Keccak-256 hash function There are seven Keccak-F permutation functions, each indicated by $\texttt{Keccak}$-$f[b]$, where $b = 5\times 5\times 2^l$ is the size of the internal state of the hash function, for $0 \leq l \leq w$. The zkProver's Keccak state machine is a verifiable automisation of a Keccak-F permutation function, which amounts to an irreversible scrambling of bits of a string $\mathbf{s} \in \mathbb{Z}^2^b$, where $b = 5\times 5\times 2^6 = 1600$. The EVM utilises the Keccak-256 hash function, which is a sponge construction with capacity $c = 512$ bits, and denoted by Keccak$[512]$. That is, the Keccak-256 notation puts emphasis on the 256-bit security level, while the Keccak$[512]$ notation seeks to depict the actual capacity of 512 bits. Bitrate and capacity Although the internal state is $\mathtt{1600}$ bits, Keccak-F intakes a fixed number of bits as input, called the $\texttt{bitrate}$ (or simply, $\texttt{rate}$) and it is denoted by $\texttt{r}$. In our specific case, the bitrate $\texttt{r} = 1088$, whilst the capacity, $\texttt{c} = 512$. The size of a single output is $\texttt{r} = 1088$ bits. However, users can choose their required length by truncating the output, which is of length $\texttt{d}$, where $\texttt{d} = 1088k$, for some positive integer k. The Keccak-F permutation used in Keccak $[c]$ is Keccak-$p[1600, 24]$ (see NIST SHA-3 Standard). Thus, given an input bit string $\mathtt{M}$ and a output length $\mathtt{d}$, Keccak c outputs a d-bit string following the previous sponge construction description. Keccak-F padding rule The Keccak-F permutation operates on a state of width $b = 1600$ bits (or 200 bytes) and a bitrate $$ \mathtt{r = b - c = 1600 - 512 = 1088}\ \texttt{bits } \mathtt{= 136}\ \texttt{bytes} $$ But not every input string comes with this tailored bit-length. Therefore, every input string is split into $\mathtt{1088}$-bit chunks, where padding is applied to the tail-end chunk with $\mathtt{1088}$ bits or lesser. The last ingredient we need to define in order to completely specify the hash function is the padding rule. In Keccak $[c]$, the padding $\texttt{pad101}$ is used. If we define $\mathtt{j = (-m-2) \mod{r}}$, where $\mathtt{m}$ is the length of the input in bits, then the padding we have to append to the original input message is, $$ \mathtt{P = 1 \mid\mid 0^{1088+j} \mid\mid 1}. $$!!! note It should be noted that our construction does not follow the FIPS-202 based standard (a.k.a SHA-3). According to the NIST specification, the SHA3 padding has been changed to $$ \mathtt{\text{SHA3-256}(M) = \text{Keccak}512}. $$ The difference is the additional $\mathtt{01}$ bits being appended to the original message, which were not present in the original Keccak specification. Keccak-F's internal state The $\mathtt{1088}$-bit (post-padding) chunks are provided sequentially, and one chunk at a time, into the Keccak-F permutation function, to be XOR-ed with a given initialization vector $\texttt{IV}$ or intermediate states. The capacity bits are typically initialised to zero bits and not affected by any external bits. However, instead of a plain bit-string of length $\mathtt{1600}$ bits, a state $\mathbf{s}$ in the Keccak SM is best visualised in 3D form as follows; - Each bit is imagined as a cube, - The entire $\mathtt{1600}$-bit state is thought of as a 3D array of cubes (bits): $\texttt{A}[5][5][64]$. That is, sixty-four ($5 \text{-bit})\times 5\text{-bit}$)-blocks of bits (cubes). See the below figure for a $\mathtt{1600}$-bit state array, displaying; 1. The 64-bit lane $\{[3,2,z]\}$ shown in orange, consisting of 64 bits, $\mathtt{Bit}[3,2,0]$ to $\mathtt{Bit}[3,2,63]$, and 2. The 5-bit column $\{[2,y,63]\}$ shown in green, consisting of 5 bits, $\mathtt{Bit}[2,0,63]$ to $\mathtt{Bit}[2,4,63]$. !Figure 1: Keccak's 1600-bit State as a 3D Array A bit in the state $\mathbf{s}$ can be denoted by $\texttt{Bit}[x][y][z]$ as an element of the 3D-array state, but as $\texttt{Bit}[x,y,z]$ to indicate its location in position (x,y,z) with respect to the Cartesian coordinate system. The mapping between the bits of the state $\mathbf{s}$, when written as a linear array of $\mathtt{1600}$ bits, and the bits when $\mathbf{s}$ is expressed as a 3D array, is given by, $$ \mathtt{s}[64(5y + x) + z] \mapsto \mathtt{Bit}[x,y,z] $$ So then, the bit $\mathtt{a} = \mathtt{Bit}[2,3,3]$ at coordinate $(2,3,3)$ is in fact the one-thousand-and-ninety-second bit of $\mathbf{s}$, because $64(5\cdot 3 + 2) + 3 = 64(17) + 3 = 1091$. This bit, $\mathtt{a} = \mathtt{Bit}[2,3,3]$, is represented in the above figure by the blue cube. See more examples of this correspondence in the table provided below. $$ \small \begin{array}{|l|c|c|c|c|c|} \hline \textbf{x} & \textbf{y} & \textbf{z} & \textbf{s[64x + 320y + z]} & \textbf{Bit}\textbf{[x,y,z]} & \textbf{Cube Colour in Fig.1.} \\ \hline \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{s[64\cdot 0 + 320\cdot 0 + 0]} = \mathtt{s[0]}\ }\ \ \text{ }\text{ } & \mathtt{Bit[0,0,0]}\text{ } & \text{ } & \texttt{Yellow} \\ \hline \mathtt{2} & \mathtt{2} & \mathtt{4} & \mathtt{s[64\cdot 2 + 320\cdot 2 + 4]} = \mathtt{s[772]}\ \text{ } & \mathtt{Bit[2,2,4]}\text{ }\text{ } & \texttt{Red} \\ \hline \mathtt{2} & \mathtt{3} & \mathtt{3} & \text{ } \mathtt{s[64\cdot 2 + 320\cdot 3 + 3]} = \mathtt{s[1091]}\text{ } & \mathtt{Bit[2,3,3]}\text{ }\text{ } & \texttt{Blue} \\ \hline \mathtt{0} & \mathtt{2} & \mathtt{3} & \mathtt{s[64\cdot 0 + 320\cdot 2 + 3]} = \mathtt{s[643]}\text{ }\text{ } & \mathtt{Bit[0,2,3]}\text{ }\text{ } & \texttt{Purple} \\ \hline \mathtt{2} & \mathtt{2} & \mathtt{63} & \text{ }\text{ } \mathtt{s[64\cdot 2 + 320\cdot 2 + 63]} = \mathtt{s[831]}\text{ } & \mathtt{Bit[2,2,63]} & \texttt{Green} \\ \hline \mathtt{2} & \mathtt{1} & \mathtt{63} & \text{ } & \mathtt{s[64\cdot 2 + 320\cdot 1 + 63]} = \mathtt{s[511]}\text{ } & \mathtt{Bit[2,1,63]} & \texttt{Green} \\ \hline \mathtt{2} & \mathtt{0} & \mathtt{63} & \text{ } \mathtt{s[64\cdot 2 + 320\cdot 0 + 63]} = \mathtt{s[191]} \text{ } & \mathtt{Bit[2,0,63]} & \texttt{Green} \\ \hline \mathtt{2} & \mathtt{4} & \mathtt{63} & \text{ }\text{ } \mathtt{s[64\cdot 2 + 320\cdot 4 + 63]} = \mathtt{s[1471]} & \mathtt{Bit[2,4,63]} & \texttt{Green} \\ \hline \mathtt{2} & \mathtt{3} & \mathtt{63} & \text{ }\text{ } \mathtt{s[64\cdot 2 + 320\cdot 3 + 63]} = \mathtt{s[1151]} & \mathtt{Bit[2,3,63]} & \texttt{Green} \\ \hline \end{array} $$ Consider computing the $\texttt{XOR}$ of all the 5 bits in the column $\{[2,y,63]\}$. If its bits are as given below, $$ \mathtt{Bit[2,2,63]} = 1,\ \mathtt{Bit[2,1,63]} = 1,\ \mathtt{Bit[2,0,63]} = 0,\ \mathtt{Bit[2,4,63]} = 0\ \text{ and }\ \mathtt{Bit[2,3,63]} = 1 $$ then the $\texttt{XOR}$ of the column bits, denoted by $C[2,63]$, is $C[2,63] = 1 \oplus 1 \oplus 0 \oplus 0 \oplus 1 = 1$. In our notation, a column is identified by the fixed x- and z- values. Hence the $\texttt{XOR}$ of all the 5 bits in a column is denoted by $\mathtt{C[x,z]}$. That is, $$ \mathtt{C[x,z]} = \mathtt{Bit[x,0,z]} \oplus \mathtt{Bit[x,1,z]} \oplus \mathtt{Bit[x,2,z]} \oplus \mathtt{Bit[x,3,z]} \oplus \mathtt{Bit[x,4,z]} $$ Keccak-F rounds The Keccak-F state machine runs 24 rounds, each of which is a composition of five step mappings; $\mathtt{\theta}$, $\mathtt{\rho}$, $\mathtt{\pi}$, $\mathtt{\chi}$, and $\mathtt{\iota}$, denoted by $$ \texttt{Rnd(A, ir)} = Î¹(Ï‡(Ï€(Ï(Î¸(A)))), \mathtt{ir}) $$ where $\texttt{ir}$ is the round index, and in our case, $0 \leq \texttt{ir} \leq 23$. These step mappings are individually described in the subsections below. The following table illustrates how the output state of each step mapping is relayed to the next step mapping as its input state. $$ \small \begin{array}{|l|c|c|c|c|c|} \hline \textbf{Pseudo-code of the Composition} & \textbf{The keccak\f.cpp Code}\\ \hline \text{Run a for-loop over bits in a 25-bit slice of the state } \texttt{S} & \texttt{for} \mathtt{(uint64\t\ ir=0; ir<24; ir++)}\\ \hline \text{ } &\ \text{ }\\ \hline \text{Apply step mapping } \theta \text{ on current state}\texttt{ S} &\text{ } \mathtt{KeccakTheta(S, ir);} \text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ } \\ \hline \text{Copy output of } \theta \text{ and set it as an input state of } \rho & \text{ } \mathtt{S.copySoutRefsToSinRefs();} \\ \hline \text{Apply step mapping } \rho \text{ to the input state. } & \text{ } \mathtt{KeccakRho(S);} \text{ } \\ \hline \text{Copy output of } \rho \text{ and set it as an input state of } \pi & \text{ } \mathtt{S.copySoutRefsToSinRefs();} \\ \hline \text{Apply step mapping } \pi \text{ to the input state.} & \text{ } \mathtt{KeccakPi(S);} \text{ } \\ \hline \text{Copy output of } \pi \text{ and set it as an input state of } \chi & \text{ } \mathtt{S.copySoutRefsToSinRefs();} \\ \hline \text{Apply step mapping } \chi \text{ to the input state.} & \text{ } \mathtt{KeccakChi(S);} \text{ } \\ \hline \text{Copy output of } \chi \text{ and set it as an input state of } \iota & \text{ } \mathtt{S.copySoutRefsToSinRefs();} \\ \hline \text{Apply step mapping } \iota \text{ to the input state.} & \text{ } \mathtt{KeccakIota(S);} \text{ } \\ \hline \text{Set output of } \iota \text{ as new state, and reset to } 0 \text{ for next round } & \text{ } \mathtt{S.copySoutRefsToSinRefs();} \\ \hline \text{ } &\} \text{ }\\ \hline \end{array} $$ Therefore, given a state value, the next value of the bit $\text{A}[x,y,z]$ is denoted by $\text{A}'[x,y,z]$. The code shown on the right side of the table is keccakf.cpp. Mapping Theta The first step mapping, $\mathtt{\theta}$, referred to as $\texttt{KeccakTheta()}$ in keccakf.cpp, can be described in three sub-steps; 1. Compute the $\mathtt{XOR}$ of the bits in the $[(x-1)\text{ mod }5, z]$-column, $$ \begin{aligned} \mathtt{C[(x-1)\text{ mod }5, z] = } \text{ } \mathtt{Bit[(x-1)\text{ mod }5,0,z]} \oplus\\ \mathtt{Bit[(x-1)\text{ mod }5,1,z]} \oplus\\ \mathtt{Bit[(x-1)\text{ mod }5,2,z]} \oplus\\ \mathtt{Bit[(x-1)\text{ mod }5,3,z]} \oplus\\ \mathtt{Bit[(x-1)\text{ mod }5,0,z]}\text{ }\text{ }\text{ } \end{aligned} $$ and the $\mathtt{XOR}$ of the bits in the $[(x+1)\text{ mod }5, (z-1)\text{ mod }64]$-column, $$ \begin{aligned} \mathtt{C[(x+1)\text{ mod }5, (z-1)\text{ mod } 64] =\text{ }} \mathtt{Bit[(x+1)\text{ mod }5,0,(z-1)\text{ mod } 64]} \oplus \\ \mathtt{Bit[(x+1)\text{ mod }5,1,(z-1)\text{ mod } 64]}\text{ }\oplus\\ \mathtt{Bit[(x+1)\text{ mod }5,2,(z-1)\text{ mod } 64]}\text{ } \oplus\\ \mathtt{Bit[(x+1)\text{ mod }5,3,(z-1)\text{ mod } 64]}\text{ }\oplus\\ \mathtt{Bit[(x+1)\text{ mod }5,0,(z-1)\text{ mod } 64]}\quad \end{aligned} $$ 2. Calculate the $\mathtt{XOR}$ of the two column $\mathtt{XOR}$s; $$ \mathtt{D[x,z]\text{ } =}\text{ } \mathtt{C[(x-1)\text{ mod }5, z]\text{ } \oplus\text{ } {C}[(x+1)\text{ mod }5, (z-1)\text{ mod } 64]} $$ 3. Compute the $\mathtt{XOR}$ of $\mathtt{D[x,z]}$ and the bit $\mathtt{A[x,y,z]}$ of the current state value; $$ \mathtt{A'[x,y,z] = {A}[x,y,z] \oplus {D}[x,z]} $$ See the below figure, for an illustration of the θ step mapping applied on one bit. The diagram is taken from Keccak Reference 3.0 guide. The code of the θ step mapping is found here: keccaktheta.cpp !Theta Step Mapping On One Bit Mapping Rho The step mapping ρ does not change the value of the input bit, but simply moves it to another position along the z-axis. Since all operations along the z-axis are worked out modulo 64, the mapping ρ is therefore cyclic, and amounts to rotating each of the 64 bits in the same lane along the z-axis. It does this in three sub-steps; 1. Set $\mathtt{(x,y) = (0,1)}$, 2. For t ranging from 0 to 23, set $\mathtt{{A}'[x,y,z] = {A}\big[x,y,\big(z-(t+1)(t+2)/2 \big)\text{mod }64\big]}$, 3. Set $\mathtt{(x,y) = (y,(2x + 3y)\text{mod }5)}$. Basically, ρ modifies the z coordinate of each bit, $\mathtt{Bit}[x,y,z]$, by subtracting a specific offset constant $\mathbf{K}$ modulo 64, where $\mathbf{K} = (t+1)(t+2)/2$. See the below table for these offset constants used for rotation. $$ \begin{array}{|l|c|c|c|c|c|c|} \hline \mathbf{t} & \mathbf{K} & \mathbf{K} \textbf{ mod } \mathbf{64} & \text{} & \mathbf{t} & \mathbf{K} & \mathbf{K} \textbf{ mod } \mathbf{64} \\ \hline 0 & 1 & 1 & \text{} & 12 & 91 & 27 \\ \hline 1 & 3 & 3 & \text{} & 13 & 105 & 41 \\ \hline 2 & 6 & 6 & \text{} & 14 & 120 & 56 \\ \hline 3 & 10 & 10 & \text{} & 15 & 136 & 8 \\ \hline 4 & 15 & 15 & \text{} & 16 & 153 & 25 \\ \hline 5 & 21 & 21 & \text{} & 17 & 171 & 43 \\ \hline 6 & 28 & 28 & \text{} & 18 & 190 & 62 \\ \hline 7 & 36 & 36 & \text{} & 19 & 210 & 18 \\ \hline 8 & 45 & 45 & \text{} & 20 & 231 & 39 \\ \hline 9 & 55 & 55 & \text{} & 21 & 253 & 61 \\ \hline 10 & 66 & 2 & \text{} & 22 & 276 & 20 \\ \hline 11 & 78 & 14 & \text{} & 23 & 300 & 44 \\ \hline \end{array} $$ The 24 constants in the above table are first permuted and then set as fixed offsets corresponding to each bit of the 3D state array. $$ \begin{array}{|l|c|c|c|c|c|} \hline \texttt{y} \big{\\ } \texttt{x} & 3 & 4 & 0 & 1 & 2\\ \hline 2 & 25 & 39 & 3 & 10 & 43\\ \hline 1 & 55 & 20 & 36 & 44 & 6\\ \hline 0 & 28 & 27 & 0 & 1 & 62\\ \hline 4 & 56 & 14 & 18 & 2 & 61\\ \hline 3 & 21 & 8 & 41 & 45 & 15\\ \hline \end{array} $$ Note that all bits $\mathtt{Bit[x,y,z]}$ such that $\mathtt{x = 0}$ and $\mathtt{y = 0}$ correspond to a zero offset constant. Consequently, the origin $\mathtt{Bit}[0,0,0]$ and all the other 63 bits along the $\{\mathtt{Bit[0,0,z]}\}$ lane remain unmoved by ρ. Although the effect of ρ is a rotation of bits along the z-axis, it actually operates on each 25-bit (x,y)-slice of the state 3D-array. Hence the 25 offset constants (and not 64), including the zero offset of the origin lane $\{\mathtt{Bit[0,0,z]}\}$. Example Here's an example of how ρ maps two different lanes. 1. For all z, where $0\leq z\leq63$, the bits $\{\mathtt{Bit}[2,4,z]\}$ are always off-set by 61 and mapped as follows, $$ \rho\big(\mathtt{Bit}[2,4,z]\big) \mapsto \mathtt{Bit}[4,1,(z-61)\text{mod }64] $$ 2. Similarly, the bits $\{\mathtt{Bit}[4,1,z]\}$ are always off-set by 20 and mapped as follows, $$ \rho\big(\mathtt{Bit}[4,1,z]\big) \mapsto \mathtt{Bit}[1,1,(z-20)\text{mod }64] $$ The code of the ρ step mapping is found here: keccakrho.cpp Mapping Pi The step mapping π is literally a shuffle of the 25 bits in each (x,y)-slice. For each z, where $0\leq z\leq63$, it is defined as the mapping, $$ \pi\big(\mathtt{Bit}[x,y,z]\big) \mapsto \mathtt{Bit}[(x+3y)\text{mod }5,x,z] $$ Note that π fixes each of the bits $\{\mathtt{Bit}[0,0,z]\}$, including the bit at the origin, $\mathtt{Bit}[0,0,0]$. Also, for all bits in the 3D-array, π does not change the z-component. That is, all bits remain in their original (x,y)-slice. To this extend, it suffices to describe π in terms of the images of the (x,y) pairs alone, as in the table below. $$ \begin{array}{|l|c|c|c|c|c|c|c|c|c|} \hline \mathbf{(x,y)} & \mathbf{x+3y} & \mathbf{(x+3y)}\textbf{mod }\mathbf{5} & \mathbf{\pi(x,y)} & \text{} & \mathbf{(x,y)} & \mathbf{x+3y} & \mathbf{(x+3y)}\textbf{mod }\mathbf{5} & \mathbf{\pi(x,y)}\\ \hline (2,2) & 8 & 3 & (3,2) & \text{} & (3,4) & 15 & 0 & (0,3) \\ \hline (1,1) & 4 & 4 & (4,1) & \text{} & (1,3) & 10 & 0 & (0,1) \\ \hline (4,4) & 16 & 1 & (1,4) & \text{} & (4,2) & 10 & 0 & (0,4) \\ \hline (3,3) & 12 & 2 & (2,3) & \text{} & (2,1) & 5 & 0 & (0,2) \\ \hline (3,0) & 3 & 3 & (3,3) & \text{} & (1,2) & 7 & 2 & (2,1) \\ \hline (4,0) & 4 & 4 & (4,4) & \text{} & (2,4) & 14 & 4 & (4,2) \\ \hline (1,0) & 1 & 1 & (1,1) & \text{} & (3,1) & 6 & 1 & (1,3) \\ \hline (2,0) & 2 & 2 & (2,2) & \text{} & (4,3) & 13 & 3 & (3,4) \\ \hline (0,2) & 6 & 1 & (1,2) & \text{} & (3,2) & 9 & 4 & (4,3) \\ \hline (0,1) & 3 & 3 & (3,0) & \text{} & (4,1) & 7 & 2 & (2,4) \\ \hline (0,4) & 12 & 2 & (2,0) & \text{} & (1,4) & 13 & 3 & (3,1) \\ \hline (0,3) & 9 & 4 & (4,0) & \text{} & (2,3) & 11 & 1 & (1,2) \\ \hline \end{array} $$ Below diagram, showing the (x,y)-slices, depicts the shuffling of the bits in accordance with the above table. This figure is in fact the mapping displayed in Figure 2.3 of the [Keccak Reference 3.0 [Page 20; 2011]](https://keccak.team/files/Keccak-reference-3.0.pdf). !How Pi shuffles bits on a 25-bit (x,y)-slice The code of the π step mapping is found here: keccakpi.cpp Mapping Chi The χ step mapping is the non-linear layer of Keccak-F, and it can be thought of as a parallel application of $320 = 564$ S-boxes operating on 5-bit rows. How Chi operates on rows For a fixed $y = b$ and $z = c$, the χ step mapping takes as its input the 5-bit row, $$ \mathbf{Row}[b,c] = \Big(\mathtt{Bit}[3,b,c],\mathtt{Bit}[4,b,c],\mathtt{Bit}[0,b,c],\mathtt{Bit}[1,b,c],\mathtt{Bit}[2,b,c]\Big) $$ It then computes a non-linear combination of each bit $\mathtt{Bit}[x,b,c]$ in $\mathbf{Row}[b,c]$, with the next two consecutive bits $\mathtt{Bit}[(x+1)\text{mod }5,b,c]$ and $\mathtt{Bit}[(x+2)\text{mod }5,b,c]$ also in $\mathbf{Row}[b,c]$. In order to change the bit, $\mathtt{Bit}[x,b,c]$, 1. χ takes $\mathtt{Bit}[(x+1)\text{mod }5,b,c]$ as input to a $\texttt{NOT}$-gate. 2. Then takes the output, $\texttt{NOT}\big(\mathtt{Bit}[(x+1)\text{mod }5,b,c]\big)$, together with $\mathtt{Bit}[(x+2)\text{mod }5,b,c]$, as inputs to an $\texttt{AND}$-gate. 3. Finally, it XOR-es the output of the $\texttt{AND}$-gate with the bit being changed, $\mathtt{Bit}[x,b,c]$. That is, for each 5-bit row, $\mathbf{Row}[b,c]$, χ can be summarized as the mapping of each bit, $\mathtt{Bit}[x,b,c]$, as; $$ \chi : \mathtt{Bit}[x,b,c] \mapsto \Big(\big(\texttt{NOT}\big(\mathtt{Bit}[(x+1)\text{mod }5,b,c]\big) \big) \texttt{ AND }\big(\mathtt{Bit}[(x+2)\text{mod }5]\big)\Big)\ \oplus\ \mathtt{Bit}[x,b,c] $$ where $x \in \{3,4,0,1,2\}$. Overall, the χ step mapping can be depicted as a "circuit" of gates as in the below diagram, taken from FIPS PUB 202, August 2015 (Figure 6, Page 15). !The KeccakChi Gates Circuit The code of the χ step mapping is found here: keccakchi.cpp Mapping Iota The $\large{\iota}$ step mapping adds 64-bit round-constants, $RC[ir]$, so as to disrupt any symmetry in the computation of the round outputs. Note that ir is the round index, $0\leq ir \leq 23$. Hence, 24 distinct round-constants are required, and are denoted in array notation as, $$ \{RC[ir][z]\ |\ 0\leq ir \leq 23 \text{ and } 0\leq z \leq 63 \} $$ The round-constants are XOR-ed only to a single lane of the state, in particular, the origin lane (i.e., the 64 bits $\{\mathtt{Bit}[0,0,z]\}$ where $0\leq z \leq 63$). Iota operating the origin lane Here's how $\large{\iota}$ operates on the origin lane, $\{\mathtt{Bit}[0,0,z]\ |\ 0\leq z \leq 63\}$ Firstly, the derivation of the round-constants $RC[ir]$. As shown in the algorithm of $\large{\iota}$, the round-constant RC is initialized to 0^{w} at the beginning of each round of $\large{\iota}$. After which, 7 specific bit-places $\{2^j â€“ 1\ |\ 0 \leq j \leq 6 \}$ of the round-constant are set to the bits $\{rc[j + 7 ir]\ |\ 0 \leq j \leq 6 \}$. Meanwhile, the rest of the bits remain as zeros. $$ \begin{array}{|l|c|c|c|c|c|c|} \hline \mathtt{j} & \mathtt{2^j - 1} & \mathtt{j + 7ir}\\ \hline 0 & 0 & 0 + 7\mathtt{ir}\\ \hline 1 & 1 & 1 + 7\mathtt{ir}\\ \hline 2 & 3 & 2 + 7\mathtt{ir}\\ \hline 3 & 7 & 3 + 7\mathtt{ir}\\ \hline 4 & 15 & 4 + 7\mathtt{ir}\\ \hline 5 & 31 & 5 + 7\mathtt{ir}\\ \hline 6 & 63 & 6 + 7\mathtt{ir}\\ \hline \end{array} $$ The derivation of the 7 bits, $\{rc[j + 7 ir]\}$, is explained below. Secondly, each round of $\large{\iota}$ simply XOR-es the corresponding 64-bit round-constant to the 64-bit lane, $\{\mathtt{Bit}[0,0,z]\ |\ 0\leq z \leq 63 \}$. Since only 7 bits of the round-constant are set, each round of $\large{\iota}$ therefore alters at most 7 bits of origin lane. $$ \textbf{Algorithm: Keccak-F Iota Mapping} \text{} \\ \textbf{Input: }\text{ state array }\mathbf{ A }\text{, round index } ir \text{ } \\ \textbf{Output:}\text{ state array }\mathbf{ A'}\text{ }\text{}\\ \text{Steps:}\text{ } \\ \text{ 1. For all triples } (x, y, z) \text{ such that } 0 â‰¤ x <5, 0 â‰¤ y < 5,\\ \text{ and } 0 â‰¤ z < w, \text{ let } \mathbf{A'}[x, y, z] = \mathbf{A}[x, y, z]\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\\ \text{2. Let } RC = 0^w. \text{ } \ \text{ }\\ \text{3. For } j \text{ from } 0 \text{ to } l, \text{ let } RC[2^j â€“ 1] = rc[j + 7 ir]. \text{ }\text{ }\text{ }\text{ }\text{ }\text{

}\text{ }\text{ }\text{ }\text{ }\text{ }\text{ }\\ \text{4. For all } z \text{ such that } 0 â‰¤ z \texttt{B} ; $$ In this case, $\mathtt{inA} \ \mathtt{=}\ 1$ and $\mathtt{setB}\ \mathtt{=}\ 1$, while the rest of the selectors and setters are set to 0. Hence, $$ \texttt{op} \ \mathtt{=}\ \mathtt{A} \mathtt{inA} + \texttt{B} \texttt{inB} + \texttt{C} \texttt{inC} + \texttt{D}\texttt{inD} + \texttt{E}\texttt{inE} \\ \ \mathtt{=}\ \mathtt{A} 1 + \texttt{B} 0 + \texttt{C} 0 + \texttt{D}0 + \texttt{E}0 \ \mathtt{=}\ \mathtt{A}. \quad\text{ }\text{ }\text{ } $$ So then, $$ \texttt{B}' \ \mathtt{=}\ (\mathtt{op - B}) \mathtt{setB} + \texttt{B} \\ \phantom{\texttt{B}'} \ \mathtt{=}\ (\mathtt{A - B}) 1 + \texttt{B} = \texttt{A} $$ That is, $\texttt{B}' = \texttt{A}$. This is read as follows, "the next value stored in register $\texttt{B}$ is $\texttt{A}$". Main execution routine Each ROM instruction stipulates which registers should be included in computing OP, as well as where the resulting output values should be stored. For each register $\texttt{X}$, the Executor therefore checks whether $\texttt{inX}$ is zero or not. It selects the register $\texttt{X}$ if $\texttt{inX} \ \mathtt{\not=}\ 0$. Similarly, the Executor only stores an output value in the register $\texttt{Y}$, only if $\texttt{setY} = 1$. The main loop of the Executor looks like this: c for (i=0; i rom.MAXCNTARITHLIMIT) { proverRequest.result = ZKRSMMAINOOCARITH; } ... Counters are the essence of the rollup solution. They are mainly used for monitoring whether the executing SM has sufficient counters to complete the operation. After processing a batch, the counters are compared against their maximum limits. Execution fails if any of the counters has been exceeded. In which case the proof cannot be generated. Memory instructions The register for the Memory operation is $\texttt{mOp}$. It is set to 1 (i.e., $\mathtt{mOp = 1}$) if a Memory instruction is to be executed, or set to zero (i.e., $\mathtt{mOp = 0}$) otherwise. Another Memory register, denoted by $\texttt{mWR}$, indicates a WRITE to Memory if $\texttt{mWR = 1}$, or a READ from Memory if $\texttt{mWR = 0}$. These operations use the Context region of Memory, denoted by ctx.mem, which acts like a map taking an address ($\texttt{addr}$) as key, and an array of 8 field elements as value. - If $\texttt{mWR = 1}$, the value in the $\texttt{OP}$ register is written to ctx.mem at a specified address. The WRITE code looks like this: c if (rom.line[zkPC].mOp==1 && rom.line[zkPC].mWR==1) // Memory write { ctx.mem[addr] = op; } - If $\texttt{mWR = 0}$, the value at a specified address in ctx.mem is read and copied to the $\texttt{OP}$ register. The READ code looks like this: c if (rom.line[zkPC].mOp==1 && rom.line[zkPC].mWR==0) // Memory read { op = pols.FREE[i] = ctx.mem[addr]; } This means values stored in any of the registers; $\texttt{A}\ \texttt{0...7}$, $\texttt{B}\ \texttt{0...7}$ and $\texttt{C}\ \texttt{0...7}$; can be stored and read from ctx.mem at $\texttt{addr}$. Binary instructions Any Binary instruction, which the main SM delegates to the Binary SM, is an operation between two values. These two input values are always taken from the 256-bit registers $\texttt{A}\ \texttt{0...7}$ and $\texttt{B}\ \texttt{0...7}$. It is important to fill the $\texttt{A}\ \texttt{0...7}$ and $\texttt{B}\ \texttt{0...7}$ registers with proper data when executing preceding instructions. Binary SM executes and verifies the following operations - Addition $\texttt{ADD}$, Subtraction $\texttt{SUB}$, Less-than $\texttt{LT}$, Signed less-than $\texttt{SLT}$, Equals $\texttt{EQ}$, $\texttt{AND}$, $\texttt{OR}$, $\texttt{XOR}$ and $\texttt{NOT}$. The details on what these operations are, have been documented here. Example For instance, when the ROM instruction is to add two values, the Binary SM reads as input values whatever values stored in the $\texttt{A}\ \texttt{0...7}$ and $\texttt{B}\ \texttt{0...7}$ registers. If the resulting sum fits in the allocated 256 bits for the Binary output, it gets stored in the $\texttt{FREE}\ \texttt{0...7}$ and then injected to the $\texttt{OP}$ register. But if the resulting sum exceeds the allocated 256 bits, the Binary operation not only updates the $\texttt{OP}$ register but also the $\texttt{cntArith}$ register and the $\texttt{carry}$ register, setting $\texttt{carry}$ to 1. Subsequently, a conditional jump then gets used. Here's how the code for the Binary instruction looks like, c for (i=0; i> 256) > 0); } Storage instructions Storage is managed by calling the $\texttt{StateDB.SMT}$, which essentially refers to Storage operations; $\texttt{Set}$ and $\texttt{StateDB}\ \texttt{Get}$. As seen in the above figure, the corresponding ROM instructions are denoted by $\texttt{sWR}$ and $\texttt{sRD}$, respectively. Values can be read from the SMT if $\mathtt{sRD == 1}$, or written to the SMT if $\mathtt{sWR == 1}$. - Reading a $\texttt{value}$ from the SMT requires passing the old state root $\texttt{oldRoot}$, the key $\texttt{sKey}$ and the read $\texttt{value}$, through the $\texttt{StateDB}$ service. And the value being read is stored in the $\texttt{FREE}\ \texttt{0...7}$ register. The code for executing an $\texttt{sRD}$ looks like this: c for (i=0; ismtGet(oldRoot, pols.sKey[i], value); FREE = value; pols.cntPoseidonG += 2; } ... } - The other way around, writing a value to the SMT also requires passing; the old state root $\texttt{oldRoot}$, the key $\texttt{sKey}$ and the written $\texttt{value}$; through the $\texttt{StateDB}$ service. The written value is taken from the $\texttt{D}\ \texttt{0...7}$ register. The code for executing an $\texttt{sWR}$ looks like this: c for (i=0; ismtSet(oldRoot, pols.sKey[i], value); pols.cntPoseidonG += 2; } ... } In both the READ and the WRITE cases; 1. The input Storage key, $\texttt{sKeyI}$, is computed as the Poseidon digest of the $\texttt{C}\ \texttt{0...7}$ register. 2. And this $\texttt{sKeyI}$ is in turn used to compute the new Storage key $\texttt{sKey}$. That is, the Poseidon digest of, - $\texttt{A}\ \texttt{0...5}$, the first six registers of $\texttt{A}\ \texttt{0...7}$, which is the address (the account number), - $\texttt{B}\ \texttt{0..1[i]}$, the first two registers of $\texttt{B}\ \texttt{0...7}$, which is the account number being written to or read from, - $\texttt{sKeyI}$, the input Storage key. 3. The $\texttt{oldRoot}$ is taken from the State Root register, $\texttt{SR}\ \texttt{0...7}$. 4. The old state root $\texttt{oldRoot}$, the key $\texttt{sKey}$ and the $\texttt{value}$ are passed through the $\texttt{StateDB}$ service. 5. The PoseidonG counter, $\texttt{cntPoseidonG}$, gets incremented by 2 because the Poseidon SM is used twice in each of this operations. All these computations happen in the $\texttt{StateDB.SMT}$, the SMT class of the $\texttt{StateDB}$. Then, all the inputs and all the outputs of this operations, go to the Storage SM for proving Computational Integrity (CI) and verification. Conclusion ROM instructions related to other secondary state machines can be described in a similar manner we have done with the Memory, Binary and Storage instructions, yet with their own special registers. All-in-all, the main SM is an Algebraic Processor, achieving efficiency through delegation. It is the major component of the zkEVM's Prover that emulates the EVM in how transactions are processed and smart contracts are deployed. The major differences are seen in the zkEVM, - executing and rolling up hundreds of transactions into one batch, and - producing easy-to-verify proofs of Computational Integrity. # circom-in-zkprover.md: Although CIRCOM is mainly used to convert a STARK proof into its respective Arithmetic circuit, it is implemented in varied ways within the zkProver, specifically during the STARK recursion process. !!!info This document gives further elaboration on the Proving architecture by focusing on the CIRCOM Arithmetic circuits, while highlighting CIRCOM's pivotal role in the STARK Recursion process. zkEVM batch prover The first step in proving computational integrity starts with the zkEVM batch prover. It is a circuit that proves correctness of the state transition from an oldStateRoot to a newStateRoot. The zkEVM batch prover takes as inputs; the oldStateRoot, oldAccInputHash, oldBatchNum, chainID and forkID. And its outputs are; the newStateRoot, newAccInputHash, localExitRoot, and newBatchNum, !zkEVM Batch Prover The localExitRoot is the Merkle root of the ExitTree used in the bridge to transfer information from the L2 to the Ethereum L1. The oldBatchNum indicates which batch is being processed, and in each transition the numbering increments by 1, such that newBatchNum = oldBatchNum + 1. All transactions being processed are encoded into the batch via the oldAccInputHash, read "old accumulated input hash". Although built in a smart contract, the encoding of transactions resembles a blockchain, where transactions are chained together with the Keccak hash function. See figure below depicting this encoding. !Transactions chained into a batch The main idea, when proving validity of a batch, is for the oldAccInputHash and its corresponding newAccInputHash to match accordingly. The resultant STARK proof denoted by $\mathtt{\pi{batch}}$, which involves 669 committed polynomials each of degree 2^{23}, is 1.9 Megabytes big. Such a validity proof is not ideal in view of its storage cost in Ethereum. The Polygon zkEVM's strategy to reducing the size of the ultimate validity proof is by implementing a recursive aggregation of many batch proofs into one. CIRCOM circuits As outlined in the Proving architecture subsection, the recursive aggregation of many STARK proofs (aimed at reducing the size of the validity proof) is accomplished in five stages; 1. Compression stage using the c12a CIRCOM circuit, 2. Normalization stage using the $\mathtt{recursive1}\ \mathtt{prover}$ CIRCOM template, 3. Aggregation stage using the $\mathtt{recursive2}\ \mathtt{prover}$ CIRCOM template, 4. Final stage using the $\mathtt{recursivef}\ \mathtt{prover}$ CIRCOM template, 5. SNARK stage using the $\mathtt{final}\ \mathtt{prover}$ CIRCOM template. In terms of code, the three CIRCOM templates used in the middle stages (Normalization stage, Aggregation stage and Final stage) are identical except for their public inputs which are; the root constants RootC and the respective input proofs. As previously mentioned, - The $\mathtt{recursive1}\ \mathtt{prover}$ only takes $\mathtt{\pi{\texttt{c12a}}}$-type proofs as input, - The $\mathtt{recursive2}\ \mathtt{prover}$ only takes any pair of $\mathtt{\pi{\texttt{rec1}}}$-type or $\mathtt{\pi{\texttt{rec2}}}$-type proofs as input, - The $\mathtt{recursivef}\ \mathtt{prover}$ only takes a pair of $\mathtt{\pi{\texttt{rec2}}}$-type proofs as input. Since the underlying proof scheme is PIL-STARK, there is a built-in $\mathtt{Verifier}$ CIRCOM circuit in each of these prover circuits. Therefore, for any index $\mathtt{x} \in \{1,2, f\}$, the proof $\mathtt{\pi{x}}$ output by any of the $\mathtt{recursivex}\ \mathtt{provers}$, is in fact a verified proof because PIL-STARK automatically verifies it. A typical $\mathtt{recursivex}\ \mathtt{verifier}$ CIRCOM template, for any $\mathtt{x} \in \{1, 2, f\}$, looks like the circuit shown below. !Figure : Typical $\mathtt{recursive1}\ \mathtt{prover}$ CIRCOM template Proof-size reductions The below table displays $\text{4}:\text{1}$ reduction in proof sizes as one progresses from one stage of the STARK recursion to the next. The numbers presented below were first publicised by the Polygon zkEVM Technical Lead, Jordi Baylina, during the StarkWare Sessions held in February 2023. The video recording can be found here. $$ \begin{array}{|l|c|c|c|c|c|c|}\hline \bf{Parameter\ Description} & \texttt{zkEVM prover}& \texttt{c12a prover}& \mathtt{rec1 / rec2\ \ prover}& \texttt{recf prover} \\ \hline \texttt{Committed polynomials} & \texttt{669} & \texttt{12} & \texttt{12} & \texttt{12} \\ \hline \texttt{Permutation checks} & \texttt{18} & \texttt{0} & \texttt{0} & \texttt{0} \\ \hline \texttt{Plookups} & \texttt{29} & \texttt{0} & \texttt{0} & \texttt{0} \\ \hline \texttt{Connection checks (copy constraints)} & \texttt{2} & \texttt{1} & \texttt{1} & \texttt{1} \\ \hline \texttt{Total number of columns} & \texttt{1184} & \texttt{65} & \texttt{45} & \texttt{45} \\ \hline \texttt{Degree of polynomials (rows)} & \mathtt{n = 2^{23}} & \mathtt{n = 2^{22}} & \mathtt{n = 2^{20}} & \mathtt{n = 2^{19}} \\ \hline \texttt{Max degree of the constraint polynomial} & \texttt{3n} & \texttt{5n} & \texttt{9n} & \texttt{9n} \\ \hline \texttt{Blowup factor} & \texttt{2} & \texttt{4} & \texttt{16} & \texttt{16} \\ \hline \texttt{Proof computation time} & \texttt{129 secs} & \texttt{14 secs} & \texttt{10 secs} & \texttt{17 secs} \\ \hline \texttt{Size of the proof} & \texttt{1.9 MB} & \texttt{494 KB} & \approx \texttt{260 KB} & \approx \texttt{505 KB} \\ \hline \end{array} $$ # composition-recursion-aggregation.md: The STARK Recursion, as implemented in the zkEVM, is a specific method for combining multiple STARK proofs into a single proof. It is a process that involves; composition, recursion, and aggregation of proofs. Composition of proofs Composition of Proofs refers to using different proving systems, one after the other, so as to generate the final validity proof. In the case of zkEVM, STARK is used as the first proving system, which produces a STARK proof π{STARK} attesting to correctness of a deterministic computation. The main idea with composition is to delegate verification of the STARK proof π{STARK} to a verification circuit C. As described in the PIL-STARK section of this document, the basic verification of a STARK proof π{STARK} is performed by a Verifier entity which takes as inputs; the Proof, the Publics, and some other Verifier parameters. In this case, if the Prover can provide a proof $\pi{CIRCUIT}$, of correct execution of the verification circuit, then it should be sufficient for verifying the original STARK. As shown in the figure below, the Verifier entity just verifies the proof $\pi{CIRCUIT}$ of the STARK verification circuit. The main advantage of this composition is that $\pi{CIRCUIT}$ is smaller and faster to verify than π{STARK}. !Simple composition Recursion of proofs Recursion of proofs is carried out in two phases; setup phase and proving phase. Setup phase consists of an alternating set of two sub-processes: converting a STARK proof to a Verifier circuit and converting a Verifier circuit to a STARK proof. Prover phase is responsible for the actual recursive generation of STARK proofs. The STARK Prover uses the required inputs to generate a STARK proof, and this STARK proof, along with the requisite public values, is used as input by the next STARK Prover. Setup phase This phase takes advantage of the fact that Verifiers are much more efficient than Provers. The design idea is to create a cascade of Verifier circuits, where at each step a lot more efficiently verifiable proof is recursively created. The process therefore consists of an alternating series of two sub-processes; converting a STARK proof into a Verifier circuit, and converting a Verifier circuit into a STARK proof, denoted by S2C and C2S, respectively. !the setup phase of recursion The overall process, as depicted in the figure above, is composed of an alternating series of sub-processes; STARK-to-CIRCUIT and CIRCUIT-to-STARK. Each Verifier circuit is described in terms of its R1CS constraints by using the CIRCOM language. Circuits are used because they are suitable for computations with limited branching, and the Verifier used in the zkEVM is a computation of this type. STARK-to-CIRCUIT or S2C Sub-process Suppose the first STARK, denoted by $\text{STARK}{\text{A}}$, is described with the parameters; pil, constants and starkinfo. $\text{STARK}{\text{A}}$ is automatically translated into its Verifier circuit, denoted by $\texttt{CIRCUIT}{\texttt{A.verifier}}$. The translation from a STARK to its Verifier circuit, herein dubbed S2C as its shorthand, is performed during the setup phase. In other words, the R1CS description of the STARK Verifier circuit can be preprocessed prior to the computation of STARK proof. CIRCOM is used as an intermediate representation language for the description of circuits. More details on how the CIRCOM is utilized is discussed in the Setup S2C section. CIRCUIT-to-STARK or C2S Sub-process The circuit definition in the form of R1CS is taken and automatically translated into a new STARK definition. In this C2S sub-process, a circuit Verifier is translated into a STARK proof. That is, a new pil description, new constants and a starkinfo. This translation, is herein referred to as C2S (short for CIRCUIT-to-STARK). Following our example, the newly generated STARK is denoted as $\text{STARK}{\text{B}}$ and it is essentially a PlonKish arithmetization of $\text{STARK}{\text{A}}$ with some custom gates of the verification circuit. See the Setup C2S section for more details. Concluding the setup phase It is worth mentioning that these recursion steps can be applied as many times as desired, while taking into account the fact that each step will compress the proof, making it more efficient to be verified but at the expense of increased Prover complexity. Finally, we remark that several artifacts for generating each STARK Prover are generated during the setup phase. See the Setup S2C and Setup C2S subsections for more information about these artifacts. Proving phase The first proof is generated by providing the first STARK Prover with the proper inputs and public values. The output proof is then passed as input to the next STARK Prover, together with public inputs. This process is recursively repeated. In the below figure, it is shown how in essence a chain of recursive STARK Provers work. !Recursive Provers Notice that the final proof is actually a circuit-based proof, which is in fact a SNARK proof, specified per implementation. More details about the proving phase can be found in the Recursion Step Proof section. Aggregation In addition to Composition and Recursion, the zkProver architecture also allows for Aggregation while generating the proofs. Aggregation is a type of proof composition in which multiple valid proofs can be collated and proved to be valid by using one proof, called the Aggregated proof. Validating such an Aggregated Proof is equivalent to validating all the collated proofs. The below figure shows an example of aggregation with binary aggregators. !binary aggregation example # final-recursion-step.md: This section focuses on the final step of the proof recursion, where the last but one intermediate STARK proof, referred to as recursivef, is verified as well as the very last conversion of a STARK proof to a circuit. Setup S2C for recursivef The idea here is the same as seen before when executing a S2C: It is to generate a CIRCOM circuit that verifies $\mathtt{\pi{rec2}}$, by mimicking the FRI verification procedure. In order to achieve this, a verifier circuit recursive2.verifier.circom is generated from the previously obtained files; - The recursive2.pil file, - The recursive2.starkinfo file and - The constant roots of the previous two proofs $\mathtt{recursive2\}\ \texttt{a.verkey.constRoot}$ and $\mathtt{recursive2\b.}\ \texttt{verkey.constRoot}$; by filling the $\mathtt{stark\}\ \texttt{verifier.circom.ejs}$ template. The output CIRCOM file recursivef.circom, obtained by running a different script called genrecursivef, is in turn compiled into an R1CS recursivef.r1cs file and a witness calculator program, recursivef.witnesscal. Both these outputs are used later on, to build and fill the next execution trace. !Figure 19: Convert the recursive2 STARK to its verifier circuit called recursivef. Setup C2S for recursivef A C2S is again executed in this step. Hence a machine-like construction is obtained from the R1CS description of the verification circuit. And, this construction must be the one whose execution correctness is equivalent to the validity of the previous circuit. In this case, the R1CS description is in the file recursivef.r1cs, and the obtained construction is described by recursivef.pil. Again, a binary for all the constant polynomials recursivef.const is generated, together with the helper file recursivef.exec, which provides allocation of the witness values into their corresponding positions in the execution trace. Since all FRI-related parameters are stored in a recursive.starkstruct file, and this file is coupled with, - the recursivef.pil file as inputs to the $\mathtt{generate\starkinfo}$ service in order to generate the recursivef.starkinfo file, and - the recursivef.const as inputs to the component that builds the Merkle tree of evaluations of constant polynomials, recursivef.consttree, and its root recursivef.verkey. !Figure 20: Convert the recursivef circuit to its associated STARK. Setup S2C for final As done previously when executing a S2C, a CIRCOM circuit that verifies $\mathtt{\pi {recf}}$ is generated by mimicking the FRI verification procedure. In order to achieve this, a verifier circuit recursivef.verifier.circom is generated from the previously obtained files; - The recursivef.pil file, - The recursivef.starkinfo file and - The constant roots of the previous two proofs $\mathtt{recursivef\}\ \texttt{a.verkey.constRoot}$ and $\mathtt{recursivef\b.}\ \texttt{verkey.constRoot}$, by filling the $\mathtt{stark\}\ \texttt{verifier.circom.ejs}$ template. This verifier CIRCOM file gets imported by the final.circom circuit in order to generate the circuit being proved, using FFLONK procedure. !Convert the recursivef STARK to its verifier circuit called final.circom}$. # index.md: The Polygon zkEVM's rollup strategy is to develop a zero-knowledge Prover (zkProver) that takes a batch of many transactions, proves their validity, and publishes a minimally-sized validity proof for verification. This document provides details of how such a validity proof is created. It is a process that involves collating a number of proofs into one, using three methods; recursion, aggregation, and composition. Proving approach zkProver is the main component of Polygon zkEVM and solely responsible for proving execution correctness. Instead of using the arithmetic circuit model, the zkProver follows the state machine model. The approach therefore is to develop a state machine that allows a prover to create and submit a verifiable proof of knowledge, and anyone can take such a proof to verify it. The process that leads to achieving such a state machine-based system takes a few steps; - Modeling the deterministic computation involved as a state machine, described in the form of an Execution Trace. - Stating the equations that fully describe the state transitions of the state machine, called Arithmetic Constraints. - Using established and efficient mathematical methods to define the corresponding polynomials. - Expressing the previously stated Arithmetic Constraints into their equivalent Polynomial Identities. These Polynomial Identities are equations that can be easily tested in order to verify the Prover's claims. A Commitment Scheme is required for facilitating the proving and verification. Henceforth, in the zkProver context, a proof/verification scheme called PIL-STARK is used. Check out the documentation here for the Polygon zkEVM's commitment scheme setting. Overall process In a nutshell, a state machine's execution trace is expressed in PIL, and this expression is called the PIL Specification of the computation represented by the state machine. In the non-recursive case, a PIL specification is transformed into a verifiable STARK proof by using PIL-STARK. Subsequently, CIRCOM takes the above STARK proof as an input and generates an Arithmetic circuit and its corresponding witness. The Arithmetic circuit is expressed in terms of its equivalent Rank-1 Constraint System (R1CS), while the witness is actually a set of input, intermediate and output values of the circuit wires, satisfying the R1CS. Finally, Rapid SNARK takes the above Witness together with the STARK Verifier data and generates a SNARK proof corresponding to the previous STARK proof. The SNARK proof gets published as the validity proof of the original computation. # intermediate-recursion.md: The first step of the proof recursion, where the first STARK proof is verified, is referred to as recursive1. All intermediate steps of recursion are referred to as recursive2, while the last step is called recursivef. Setup S2C for recursive1 At this point in the recursion process, the first STARK proof π has been validated with a STARK proof $\pi\texttt{c12a}$. The idea now is to generate a CIRCOM circuit that verifies $\pi\texttt{c12a}$, by mimicking the FRI verification procedure. In order to achieve this, a verifier circuit c12a.verifier.circom is generated from the previously obtained files; - The c12a.pil file, - The c12a.starkinfo parameters, - The constant root c12a.verkey.constRoot, by filling the $\mathtt{stark\}\ \texttt{verifier.circom.ejs}$ template as before. In this case, as mentioned in the Normalization stage subsection of the Recursion section, the circuit had to be slightly modified in order to include the constant root as a public input. This is important especially for the Aggregation stage, where all computation constants dependent on the previous circuit need to be provided as public inputs. This is done by using the recursive1.circom file, and internally importing the previously generated c12a.verifier.circom circuit as a library. The verifier circuit is instantiated inside $\mathtt{recursive1.circom}$, connecting all the necessary wires and including the constant root to the set of publics. !Convert the c12a STARK to a c12a verifier circuit The output circom file recursive1.circom is compiled into a R1CS recursive1.r1cs file and a witness calculator program, $\mathtt{recursive1.witnesscal}$, which is used for both building and filling the next execution trace. Setup C2S for recursive1 As seen previously, a machine-like construction, whose correct execution is equivalent to the validity of the previous circuit, is obtained from the R1CS description of the verification circuit. In this case, the R1CS description is in the file recursive1.r1cs, and the obtained construction is described by recursive1.pil. Again, a binary for all the constant polynomials recursive1.const is generated, together with the helper file recursive1.exec, which provides allocation of the witness values into their corresponding positions in the execution trace. Note that all FRI-related parameters are stored in a recursive.starkstruct file,located in the prover repository, and it is coupled with the following, - The recursive1.pil file as inputs to the $\mathtt{generate\starkinfo}$ service in order to generate the recursive1.starkinfo file. - The recursive1.const as inputs to the component that builds the Merkle tree of evaluations of constant polynomials, recursive1.consttree, and its root recursive1.verkey. In this case, a blowup factor of $2^4 = 16$ is used, and thus allowing the number of queries to be 32. !Convert the recursive1 circuit to its associated STARK Setup S2C for recursive2 As before, a CIRCOM circuit is generated that verifies $\mathtt{Ï€{rec1}}$ by imitating the FRI verification procedure. In order to do this, a verifier circuit recursive{1.verifier.circom is generated from the previously obtained files; - The recursive1.pil file, - The recursive1.starkinfo file, - The constant root recursive1.verkey.constRoot, by filling the verifier $\mathtt{stark\verifier.}\texttt{circom.ejs}$ template. Once the verifier is generated using the template, the template is used to create another CIRCOM that aggregates two verifiers. Note that, in the previous step, the constant root was passed hardcoded from an external file into the circuit. That's the very reason for having the Normalization stage: enabling the previous circuit and anyone verifying each or both proofs to have the exact same form, and thus allowing iterated recursion. Henceforth, the recursive2.circom circuit has two verifiers and two multiplexors that are actually deciding the form of each of the verifiers: - if the proof is $\mathtt{\pi{rec1}}$-type, the hardcoded constant root is input, but - if the proof is a $\mathtt{\pi{rec2}}$-type, the constant root should be connected as an input signal, coming from a previous circuit. A schema of the recursive2 circuit generated is as shown in the below Figure. !Figure 16: Convert the recursive1 circuit to its associated STARK Observe that, since the upper proof is of the $\mathtt{\pi{rec2}}$-type, the multiplexor does not provide the constant root rootC to the Verifier A for hardcoding it, because this verifier should get it through a public input from the previous circuit. Otherwise, since the lower proof has the $\mathtt{\pi{rec1}}$-type, the Multiplexor lets it pass through by providing the constant root to the Verifier B, so that it can be hardcoded when the corresponding template is filled. The output CIRCOM file recursive2.circom , is obtained by running a different script called genrecursive which is compiled into an R1CS recursive2.r1cs file and a witness calculator program recursive2.witnesscal and they are both used, later on, to build and fill the next execution trace. !Convert the recursive1 STARK to its verifier circuit called recursive2 Setup C2S for recursive2 As seen before, when executing a C2S, a machine-like construction gets obtained from the R1CS description of the verification circuit. This construction is specifically the one whose execution correctness is equivalent to the validity of the previous circuit. And it is described by a PIL recursive2.pil file. The R1CS description taken as input to produce this construction is in the file recursive2.r1cs. The other outputs of the recursive2 setup component are; - A binary for all the constant polynomials recursive2.const, and - The helper file recursive2.exec, which provides allocation of the witness values into their corresponding positions in the execution trace. Note that all the FRI-related parameters are stored in a recursive.starkstruct file, and in the next step, it is paired up with, - the recursive1.const as inputs to the component that builds the Merkle tree of evaluations of constant polynomials, recursive2.consttree and its root recursive2.verkey. - the recursive2.pil file as inputs to the $\mathtt{generate\starkinfo}$ service in order to generate the recursive1.starkinfo file. In this case, we are using the same blowup factor of $2^4 = 16$, allowing the number of queries to be 32. !Convert the recursive2 circuit to its associated STARK. # proof-generation-phase.md: This document explains the proof generation phase for all proofs; the zkEVM STARK, the compression c12a step, the recursion proof recursion1, the intermediate recursion proof recursion2 and the final recursion proof recursionf. Proof of the zkEVM STARK The execution trace has up to this point been built, together with a PIL file describing the ROM of the zkEVM. Given these two, a STARK proof which attests to the correct execution of the zkEVM, can be generated using the PIL-STARK tooling explained here. In this step, a blowup factor of 2 is used, so the proof becomes quite big due to a huge amount of polynomials. The compression step c12a was added for this very reason, raising the blowup factor and thus reducing the number of polynomials. !Generation for a zkEVM Proof. In order to generate the proof, the $\mathtt{main\prover}$ service is used, and requires as input; - The execution trace (that is, the committed and constant polynomials files generated by the executor using the PILCOM package), - The constant tree binary file in order to be hashed to compute the constant root, - The PIL file of the zkEVM ROM, zkevm.pil, - All the information provided by the zkevm.starkinfo.json file, including all the FRI-related parameters such as the blowup factor or the configuration of the steps. This step is intended to start the recursion, and therefore differs from the subsequent ones. However, aiming at uniformity of the code, the Main Prover procedure chooses to abstract the notion of proving. And it is intended to be the same at each step of the recursion. Proof of c12a In order to generate the proof verifying the previous zkevm.proof, all the witness values can be generated and mapped correctly into its corresponding position of the execution trace in the same exact manner as before, obtaining a binary file c12a.commit for the committed polynomials of the execution trace. Having the execution trace (that is, the committed and constant polynomials filled) and the PIL, a proof validating the previous big STARK proof can be generated. The same $\mathtt{main\prover}$ service used earlier is again used here, and as before it takes as input the previously built constant tree c12a.constTree and the c12a.starkinfo file. It in turn generates the proof c12a.proof and the publics c12a.public combined in the c12a.zkin.proof file. !Generate a STARK proof for c12a. Proof of recursive1 In order to generate the proof that verifies the previous c12a.proof, all the witness values are generated and mapped correctly into their corresponding positions of the execution trace in the exact same way as before, obtaining a binary file recursive1.commit for the committed polynomials of the execution trace. Having the execution trace (that is, the committed and constant polynomials filled) and the PIL, a proof validating the previous big STARK proof can be generated. The same $\mathtt{main\prover}$ service used previously is applied again here, it again takes as input the previously built constant tree recursive1.constTree and the recursive1.starkinfo file. This generates the proof and the publics included in the recursive1.zkin.proof file. !Generate a STARK proof for recursive1. Proof of recursive2 To generate the proof verifying the previous recursive1.proof, all witness values must be generated and mapped correctly into their corresponding positions of the execution trace in the exact same way as before, obtaining a binary file recursive2.commit for the committed polynomials of the execution trace. Having the execution trace (that is, the committed and constant polynomials filled) and the PIL, a proof validating the previous big STARK proof can generated. The same service $\mathtt{main\prover}$ generates this proof, as it was done before, it takes as inputs the previously built constant tree recursive2.constTree and the recursive2.starkinfo file. This generate the proof and the publics combined in the recursive2.zkin.proof file. !Generate a STARK proof for recursive2. Proof of recursivef To generate the proof verifying the previous recursive2.proof, we generate all the witness values and map them correctly into its corresponding position of the execution trace exactly in the same way as before, obtaining a binary file recursivef.commit for the committed polynomials of the execution trace. Having the execution trace (that is, the committed and constant polynomials filled) and the PIL, we can generate a proof validating the previous big STARK proof. Again, the same $\mathtt{main\prover}$ service is used to generate the proof. As before, it takes as inputs the previously built constant tree $\mathtt{recursivef.constTree}$ and the $\mathtt{recursivef.starkinfo}$ file. This generates the proof and the publics included in the recursivef.zkin.proof file. !Generate a STARK proof for recursivef. Proof of the final Stage The last circuit, final.circom is the one used to generate the proof. At this moment a FFLONK proof is generated. Remarks The setup phase runs with the proverjs. The proof generation runs with the prover written in C. The circuits build in the setup phase can be used as many times as desired. The Prover receives the information about the particular composition of proofs with an RPC API. # proving-architecture.md: Focusing specifically on the proving phase of the recursion process; it is a process that starts with proofs of batches (these are sequenced batches of transactions) and culminates in a ready-to-be-published validity proof, which is a SNARK proof. There are five intermediate stages to achieving this; the Compression stage, the Normalization stage, the Aggregation stage, the Final stage and the SNARK stage. An overview of the overall process can be seen in the below figure. !Proving architecture with recursion, aggregation and composition Compression stage Recall that the first STARK, in the sequence of Recursive provers seen in the Proving phase subsection of the Recursion section, generates a big proof because of its many polynomials, and its attached FRI uses a low blow-up factor. Henceforth, in each proof of batches, a compression stage is invoked, aiming at reducing the number of polynomials used. This allows the blow-up factor to be augmented, and thus reduce the proof size. A component called the $\mathtt{c12a\ prover}$ is utilized in this stage which takes a batch proof $\mathtt{{\pi}{batch}}$ as input and outputs a 'compressed' proof, denoted by $\mathtt{{\pi}{\texttt{c12a}}}$ in the above figure. Normalization stage Following completion of the compression stage, is the normalization stage. Each output of the $\mathtt{c12a\ prover}$ is taken as an input to the $\mathtt{recursive1}$ $\mathtt{prover}$ circuit. Outputs of this circuit are referred to as $\mathtt{\pi{rec1}}$-type proofs. It is in the next stage, called the Aggregation stage, which is in charge of joining several batch proofs into a single proof that validates each of the single input proofs all at once. The way to proceed is to construct a binary tree of proofs, where a pair of proofs is proved one pair at a time. However, since the aggregation of two proofs requires the constant root of the previous circuits, through a public input coming from the previous circuit, the Normalization stage is basically created for this very purpose. The stage is in charge of transforming the obtained verifier circuit, that validates the $\pi{\texttt{c12a}}$ proof, into a circuit that makes the constant root public to the next circuit. This step allows each aggregator verifier and the normalization verifier to be exactly the same, permitting successful aggregation via recursion. Aggregation stage Once the normalization step has been completed, the next stage is the aggregation of proofs (i.e., normalized proofs). In this stage, two normalized proofs are joined together by a $\mathtt{recursive2}$ $\mathtt{prover}$ component. In order to achieve this, a circuit capable of aggregating two verifiers is created, call it the $\mathtt{recursive2\ prover}$ circuit. Its outputs are proofs of the $\mathtt{\pi{rec2}}$-type. This $\mathtt{recursive2\ prover}$ circuit is repeatedly applied to pairs of proofs until there are no more normalized proofs to be aggregated. However, as observed in the figure above, the inputs to the $\mathtt{recursive2}$ $\mathtt{prover}$ can be proofs of either the $\mathtt{\pi{rec1}}$-type or the $\mathtt{\pi{rec2}}$-type. This allows us to aggregate a pair of $\mathtt{\pi{rec1}}$-type proofs, or a pair of $\mathtt{\pi{rec2}}$-type proofs, or even a combination of a $\mathtt{\pi{rec1}}$-type proof and a $\mathtt{\pi{rec2}}$-type proof. Final stage The final stage is the very last STARK step during the recursion process, and it is in charge of verifying a $\mathtt{{\pi}{rec2}}$ proof over a completely different finite field, the one defined by the $\text{BN}128$ elliptic curve. More specifically, the hash used in generating the transcript works over the field of the $\text{BN}128$ elliptic curve. Hence, all the challenges (and so, all polynomials) belong to this new field. The reason for the change to the $\text{BN}128$ elliptic curve is because a $\texttt{FFLONK}$ SNARK proof, which works over this type of elliptic curves, is to be generated in the next step of the process. This step is very much similar to the $\mathtt{recursive2\ prover}$ circuit. It instantiates a verifier circuit for $\mathtt{{\pi}{rec2}}$ except that, in this case, 2 constant roots should be provided (a constant for each of the proofs aggregated in the former step). SNARK stage The last step of the whole process is called the SNARK stage, and its purpose is to produce a $\texttt{FFLONK}$ proof $\mathtt{{\pi}{FFLONK}}$ which validates the previous $\mathtt{{\pi}{recf}}$ proof. In fact, $\texttt{FFLONK}$ can be replaced with any other SNARKs. One alternative SNARK which was previously used is $\texttt{Groth16}$, which requires a trusted setup for every new circuit. A SNARK is chosen to replace a STARK with the aim to reduce both verification complexity and proof size. SNARKs, unlike STARK proofs, have constant complexity. The $\mathtt{{\pi}{FFLONK}}$ proof gets published on-chain as the validity proof. The verifier smart contract living on the L1 verifies the validity proof. Remark on inputs All public inputs used throughout the entire recursion procedure get hashed together, and the resulting digest forms the public input to the SNARK circuit. The set of all public inputs is listed below. - oldStateRoot - oldAccInputHash - oldBatchNum - chainId - midStateRoot - midAccInputHash - midBatchNum - newStateRoot - newAccInputHash - localExitRoot - newBatchNum # proving-setup-phase.md: All preprocessing happens in the Setup phase. This means all artifacts needed for generating proofs are created in this phase. This includes the generation of intermediate circuits, which are a finite set of circuits that allow arbitrary combinations of proof recursions and aggregations. Building the zkEVM STARK The first step in building the zkEVM STARK is to build the ROM of the zkEVM state machine, where this ROM is nothing but a program containing instructions for the executor to generate a specified execution trace of the zkEVM. And it is written in JSON as rom.json. The PIL code, zkevm.pil, is built for validating the execution trace. The executor uses both the rom.json and zkevm.pil to generate all the constant polynomials for the execution trace of the zkEVM, rom.const. Observe that, as previously mentioned, committed polynomials are not needed in the setup phase, so at this stage there is no need to run the executor of the zkEVM in order to generate them. See the below schematic diagram of the process used when building a zkEVM STARK. !Build the zkEVM STARK Next to be built is the Merkle tree of evaluations of the constant polynomials, zkevm.consttree. The root of this Merkle tree, which is a hash that serves as a cryptographic fingerprint of all the fixed parameters of the computation, is stored as a parameter in a file called zkevm.verkey. The last piece of data that needs to be generated, before building the STARK, is the starkInfo necessary for automatically generating the circuit that verifies the zkEVM STARK. In this case, a blowup factor of 2 and 128 queries are used to generate the proof. The artifacts marked in gray, in the above figure, are those being used in generating the proof. Further delineation of the proof generation is provided in later sections. Setup S2C for zkEVM STARK The next step in the setup phase is to generate the circuit that verifies the zkEVM STARK (see the below Figure). !Converting the zkEVM STARK verification into a circuit The pil2circom process fills a CIRCOM EJS template, called $\mathtt{stark\verifier.circom.ejs}$, with all the necessary information needed to validate the zkEVM STARK. We henceforth need to add the zkevm.pil in order to capture - polynomial names, - the zkevm.starkinfo file which specifies the blowup factor, - the number of queries, - the steps of the FRI-verification procedure, - the constRoot in the zkevm.verkey file, and to automatically generate a circuit in CIRCOM. The CIRCOM output file zkevm.verifier.circom is then compiled into R1CS constraint system, written in a file called zkevm.verifier.r1cs. These constraints are used in the next step to generate the PIL code and the constant polynomials for the next proof. On the other hand, the CIRCOM compilation also outputs a witness calculator program called zkevm.verifier.witnesscalc. As it can be observed in the picture, the witness calculator program is marked in gray because it is used when the proof is generated. Since the aim in the next proof generation is compression (that is, proof size reduction), a blowup factor of 4 is used in this step, with 64 queries. This information is contained in the c12a.starkstruct file located in the proverjs repository. Setup c12a The zkEVM STARK is verified by a circuit called zkevm.verifier. This is the c12a circuit previously seen in the compression stage, at the beginning of the Proving Architecture. It is so called because the PIL code that verifies the c12a circuit, is a PlonKish circuit with custom gates and 12 polynomials, aiming at compression. !Convert the zkEVM verifier circuit to a STARK called c12a Given the above-mentioned R1CS description of the verification circuit zkevm.verifier.r1cs, a machine-like construction whose correct execution is equivalent to the validity of the previous circuit is obtained. This construction is described in the PIL file, c12a.pil. The process starts through a service called $\mathtt{compressor12\setup}$ (which is part of the c12a setup component, seen in the above figure), where the corresponding PIL file c12a.pil for verifying the trace is an output, together with a binary c12a.const for all the constant polynomials. Moreover, a helper file called c12a.exec is generated by the same service. This helper file contains all the necessary rules that allow the shuffling of all the witness values, which are computed later on, into the corresponding position of the execution trace. The design of this shuffling, together with the connections defined in the constants polynomials c12a.const ensures that, for an honest prover, this newly generated trace is valid whenever the previous circuit is valid. # proving-tools.md: In this document, we provide a brief outline of the proving tool called PIL-STARK and the two proving techniques namely FRI and STARK. PIL-STARK PIL-STARK proof/verification consists of three components; Setup, Prover, and Verifier. - Setup refers to the preprocessing of all data required by the Prover and Verifier. It takes the PIL description of the State Machine being proved and verified as well as the STARK configuration JSON-file as inputs. - Prover requires some input values in the form of a JSON-file together with the evaluations of the constant polynomials, the constTree and the starkInfo from the preprocessing. It outputs the STARK proof and public values called publics. - The Verifier receives the STARK proof and public values from the Prover, as well as the starkInfo and constRoot from the Setup phase. The Verifier's output is either an Accept if the proof is accepted, or a Reject if the proof is rejected. The full details of the PIL-STARK process are given here, while the following diagram summarizes entire PIL-STARK process. !PIL-STARK Process Role of FRI The zkProver uses an extended version of the STARK protocol, dubbed eSTARK, which is especially designed to prove PIL specifications. It is so called because it is not confined to proving polynomials equalities, but extends to arguments such as lookups, permutations or even copy-constraints (called connection arguments). The eSTARK protocol is composed of two main phases; the low-degree reduction phase and the FRI phase. !!!info What is FRI FRI refers to Fast Reed-Solomon Interactive Oracle Proof of Proximity. The FRI protocol consists of two phases: a commit phase and query phase. You can read more about the protocol in this document by StarkWare Team. 1. Low-degree reduction phase During this phase, we receive a FRI polynomial, which codifies the validity of the execution trace values according to the PIL code into the fact that it has a low degree. This polynomial, along with numerous other polynomials required to provide consistency checks, is committed to the Verifier. 2. FRI phase Following the acquisition of the FRI polynomial, the Prover and Verifier communicate using the standard FRI Protocol, with the goal of precisely demonstrating and verifying that the committed polynomial has a low degree. More specifically, it demonstrates that the committed values of the polynomials raise a function that is sufficiently near to a low degree polynomial. eSTARK protocol On a high level, the description of the eSTARK protocol can be broken down into several rounds. Here's what each rounds aims to achieve: - $Round\ 1$: Given the trace column polynomials after interpolating the execution trace, the Prover commits to these polynomials. - $Round\ 2$: For each lookup argument, the Prover commits to the so-called h-polynomials of the modified Plookup version described in PlonKup. - $Round\ 3$: The Prover commits to the grand-product polynomials for each of the arguments appearing in the PIL code, together with some intermediate polynomials used to reduce the degree of the grand products. This is due to the fact that PIL-STARK imposes a degree bound when committing to a polynomial. See Plookup or Plonk for the specification of the grand-products of each of the different arguments allowed in PIL. - $Round\ 4$: The Prover commits to two polynomials $Q1$ and $Q2$ arising from the splitting of the quotient polynomial Q. - $Round\ 5$: The Prover provides the Verifier with all the necessary evaluations of the polynomials so that corresponding checks can be executed. - $Round\ 6$: The Prover receives two random values from the Verifier, which are used to construct the FRI polynomial. Then the Prover and the Verifier get engaged (non-Interactively) in a FRI Protocol, which ends with the Prover sending the corresponding FRI proof to the Verifier. After the proof is generated, it is sent to the Verifier instance for the verification procedure to begin. The final output is either an accept or a reject, indicating whether the proof was accepted or rejected. # recursion-sub-process.md: This document provides a deep dive into the sub-processes S2C and C2S of the Recursion of Proofs. Setup S2C Recall that S2C denotes the process of converting a given STARK into its verifier circuit, which is a description in CIRCOM, compiled into the corresponding R1CS constraints. The architecture of this generic conversion is depicted in the below figure, where a $\mathtt{STARKx}$ is converted into a circuit denoted as $\textbf{C}{\text{y}}$. !Detailing the Setup S2C The input of S2C is all the information needed to set up a circuit for verifying the given STARK. In our architecture, the inputs are; - The PIL file x.pil, specifying STARK constraints that are going to be validated and the polynomial names, - The x.starkInfo file containing the FRI-related parameters (blowup factor, the number of queries to be done, etc.), and - The x.verkey which is the root (constRoot) of the Merkle tree of the computation constants. The output y.circom of the generate circom process, is a CIRCOM description. The circuit is actually generated by filling an EJS template for the CIRCOM description using the constraints defined by the PIL file, the FRI-related parameters included in the starkInfo file and the constRoot. As illustrated in the below figure, the inputs of the generated STARK verifier circuits are divided in two groups; private inputs and public inputs called publics. !Inputs of the STARK verifier circuits The private inputs are the parameters of the previous STARK proof: - rootC: Four field elements representing the root of the Merkle tree for the evaluations of constant polynomials (that is, preprocessed polynomials) of the previous STARK. In some of the intermediate circuits that we generate, rootC is an input of the circuit, while in other generated circuits rootC are internal signals hardcoded to the *corresponding values. (More information for each particular case is provided later in the documentation). - root1: Four field elements representing the root of the Merkle tree for the evaluations of all the trace column polynomials of the execution trace. - root2: Four field elements representing the root of the Merkle tree for the evaluations of the h-polynomials appearing in each lookup argument of the previous STARK. This root may be 0 if no lookup argument is provided in the PIL. - root3: Four field elements representing the root of the Merkle tree for the evaluations of the grand product polynomials appearing in each argument (that is, lookup, permutation or connection arguments) of the previous STARK, and the intermediate polynomials appearing in certain splitting of them. This root may be 0 if no arguments are provided in the PIL. - root4: Four field elements representing the root of the Merkle tree for the evaluations of the splitting $Q1$ and $Q2$ of the Q polynomial of the previous STARK. - evals: Contains all the necessary evaluations for all the polynomials appearing in the FRI verification process at a challenge value z and at gz. - siroot: Four field elements representing the root of the Merkle tree for the evaluations of the i-folded FRI polynomial. That is, the polynomial appearing in the i-th step of the FRI verification. - sivals: The leavesâ€™ values of the previous Merkle tree used to check all the queries. The total amount of such values depends on the number of queries and the reduction factor attached to the current step of the FRI. - sisiblings: Merkle proofs for each of the previous evaluations. - finalPol: Contains all the evaluations of the last stepâ€™s folding polynomial constructed in the FRI verification procedure, over the last defined domain, which has the same size as the degree of the polynomial. The publics are a set of inputs used by the verifier to check the final proof, and also by the intermediate STARKs. More information about publics used in the zkEVM STARK is provided in later sections of this document. In order to complete the S2C step, the final process is compiling a STARK into its CIRCOM description, so as to obtain a file with the associated R1CS constraints and a witness calculator program capable of computing all the values of the circuit wires for a given set of inputs. Finally, we remark that the particular intermediate circuit generated in a S2C step, denoted as $\textbf{C}{\text{y}}$ as seen in the S2C figure above, can be just a verification of the previous $\mathtt{STARKx}$, if we are applying only one recursion step. But more generally, other types of circuits, including the verifier, can be used to provide more functionality. This latter case is applied when circuits are used to verify aggregation of proofs. Setup C2S The proving architecture is designed to create a chain of STARKs. Given a STARK proof, the S2C step translates the proof into a verifier circuit. In the C2S step, which is part of the pre-processing, the verifier circuit is translated into a STARK proof. A picture of a generic C2S step is displayed in the below Figure, where a circuit denoted by $\textbf{C}{\text{y}}$ is converted into its corresponding STARK, denoted by $\mathtt{STARK}{\text{y}}$. !Setup recursion step C2S As shown in the above figure, the first process of the C2S step is the PIL setup, which takes the R1CS constraints of a given intermediate circuit as input, and produces all the STARK-related artifacts. These includes the associated STARK identity constraints and the computation constants that are respectively stored in a PIL file (y.pil) and in a file of constants (y.const). In particular, the identity constraints of the PlonKish arithmetization are generated by filling an EJS template for the associated PIL (for the Polygon zkEVM, the template used is called compressor12.pil.ejs). The PIL setup also generates an important executable file (y.exec) which defines how to rearrange the values produced by the circuit witness calculator into the appropriate values of the STARK Execution Trace. Note that the rearrangement rules and the computation constants only depend on the circuit shape, which is encoded in the .r1cs file generated by the CIRCOM compiler. In other words, these parameters do not depend on the particular values of the circuit wires computed for a particular input. Nevertheless, we will later on use the rearrangement rules file together with the witness values for a given input to build the STARK Execution Trace, which in turn is needed to generate the STARK proof. Finally, we also produce the starkInfo file and a Merkle tree with the STARK constants. Arithmetization In more detail, the STARK arithmetization of the intermediate circuits in our proving architecture is a PlonKish arithmetization with custom gates, using 12 polynomials for the values of the gate wires of the computation trace. The STARK arithmetization includes several other custom gates for carrying out specific tasks more efficiently. In particular, the custom gates providing the various functionalities are; - Poseidon: This custom gate is capable of validating a Poseidon hash from a given eight (8) field elements as inputs, four (4) field elements as the capacity, and a variable number of output elements. More specifically, this circuit implements the MDS matrix as its diffusion layer, and the 7-th power of field elements computations as the non-linear layer (S-boxes), in executing each of the rounds of the Poseidon hash's sponge construction. The Poseidon hash is documented here. - Extended Field Operations: This custom gate is capable of validating multiplications and additions (or a combination of the two) over the extended field $\mathbb{F}p^3$. The inputs are three (3) elements a, b and c in $\mathbb{F}p^3$ and the corresponding output is: $$ a Â· b + c â^ˆ \mathbb{F}p^3 $$ where the operations are defined over $\mathbb{F}p^3$. Observe that one can compute pure multiplications by setting c to be equal to 0. Similarly, pure additions can be computed by setting b equal to 1. - FFT: This custom gate is in charge of computing Fast Fourier Transforms (FFTs) of a variable size in $\mathbb{F}p$ or in an extension field. - Polynomial Evaluation: This custom gate is in charge of computing a single evaluation of polynomials in $\mathbb{F}p^3$ using Hornerâ€™s rule. The input consists on a field element $z \in \mathbb{F}p^3$ and the coefficients of the polynomial p which we are going to evaluate. The output is the evaluation $p(z) \in \mathbb{F}p^3$. Selector polynomials Selector polynomials are constant (or pre-processed) polynomials used to activate the custom gates of STARK. In particular, the selector polynomials used in the zkProver are; POSEIDON12, GATE, CMULADD, EVPOL4 and FFT4. The selector GATE is actually in charge of activating a basic PlonK gate. The other selectors are in charge of selecting whichever sub-circuit needs to be executed. In addition, there exists a special selector called PARTIAL, which is in charge of distinguishing between partial and full non-linear layers, when executing the Poseidon custom gate. Recursion step proof The proof of each intermediate STARK needs to be computed in order to generate the final proof. See the diagram in the Proving Phase subsection. Each intermediate STARK proof is generated using the witness values provided by the execution of the associated circuit witness calculator program, which in turn takes as inputs the publics and the values of the previous proof. These values are then rearranged properly in order to build the STARK execution trace using the corresponding .exec file. The below figure provides the schema of how the proof of an intermediate STARK is generated. !STARK Proof of a recursion step 1. Observe that the STARK executor process takes the parameters of the previous proof together with the public inputs; - the file x.zkin.proof which has the proper format for the witness calculator generated by the CIRCOM compiler - the y.pil, which is the PIL code of the current STARK - the witness calculator program (y.witnesscalc) of the associated circuit - the y.exec file containing rearrangement rules With these inputs, the STARK executor generates the non-preprocessed part of the STARK execution trace (y.commit). 2. Next, the STARK Prover process takes the execution trace. That is; the committed and constant polynomials, the constant tree, the corresponding PIL file, and the information provided by the zkevm.starkinfo.json file; in order to generate the proof. 3. Finally, when the proof is generated, the STARK Prover process generates three files: - Proof File (y.proof.json): A JSON file containing the whole STARK proof in a .json file - Publics File (y.public.json): A JSON file containing only the publics - zkIn File (y.zkin.proof.json): A JSON file combining both the proof and the publics. # construct-key-path.md: Let us first look at the zkProver's storage parameters. In the Storage SM, keys and values are strings of 256 bits. Keys are henceforth represented as 256-bit unsigned integers that are quadruples of 64-bit field elements. For example: $$ \text{Key}{\mathbf{0123}} = \big(\text{Key}{\mathbf{0}} , \text{Key}{\mathbf{1}} , \text{Key}{\mathbf{2}} , \text{Key}{\mathbf{3}} \big) $$ where each $\text{Key}{\mathbf{i}} \in \mathbb{F}p$, where $p = 2^{64} - 2^{32} + 1$. Although hashed-values are also 256-bit long and are used in quadruple form, committed values are 256-bit long and are often expressed as octets. It is mainly due to the Poseidon SM convention, where 256-bit committed values are input as $$ \text{V}{\mathbf{01..7}} = \big(\text{V}{\mathbf{0}} , \text{V}{\mathbf{1}} , \text{V}{\mathbf{2}} , \dots , \text{V}{\mathbf{7}} \big) $$ and each 32-bit $V{\mathbf{j}}$ chunk of bits. In fact, almost every other 256-bit value in storage is expressed in the form of a quadruple of 64-bit field elements. How keys are created In the key-value pair SMT context of our storage design, a key uniquely identifies a leaf. And it is because, although values can change, keys do not. Keys must consequently be generated deterministically, and in such a way that there are no collisions. That is, there must be a one-to-one correspondence between keys and leaves. A collision-resistant hash function is therefore the best tool for generating keys. And the most convenient way to generate keys is by hashing some specific information so that the resultant hash uniquely identifies the leaf. The specific information used for generating keys is the Ethereum address and some constant. The Poseidon hash function is again used for this purpose. Constructing navigation paths A path refers to the edges traversed from the root to a leaf. Since the SMTs are binary, all edges can be thought of as labeled with either a bit 0 or 1. Edges to the left are labeled with a bit 0, while edges to the right are labeled with a bit 1. Paths are therefore strings of bits, and are derived from keys in a very specific way. First of all, every key can be thought of as a quadruple, $$ \text{Key}{\mathbf{0123}} = \big(\text{Key}{\mathbf{0}} , \text{Key}{\mathbf{1}} , \text{Key}{\mathbf{2}} , \text{Key}{\mathbf{3}} \big) \in \mathbb{F}{p}^4 $$ Denote each 64-bit key part $\text{Key}{\mathbf{i}}$ bit-wise as; $$ \begin{aligned} \text{Key}{\mathbf{0}} = k{\mathbf{0,63}\ } k{\mathbf{0,62}\ } \dots k{\mathbf{0,2}\ } k{\mathbf{0,1}\ } k{\mathbf{0,0} },\\ \text{Key}{\mathbf{1}} = k{\mathbf{1,63}\ } k{\mathbf{1,62}\ } \dots k{\mathbf{1,2}\ } k{\mathbf{1,1}\ } k{\mathbf{1,0} },\\ \text{Key}{\mathbf{2}} = k{\mathbf{2,63}\ } k{\mathbf{2,62}\ } \dots k{\mathbf{2,2}\ } k{\mathbf{2,1}\ } k{\mathbf{2,0} },\\ \text{Key}{\mathbf{3}} = k{\mathbf{3,63}\ } k{\mathbf{3,62}\ } \dots k{\mathbf{3,2}\ } k{\mathbf{3,1}\ } k{\mathbf{3,0} }, \end{aligned} $$ where the most-significant bit $\text{MSB}(\text{Key}{\mathbf{i}}) = k{\mathbf{i,63}\ }$ and the least-significant bit $\text{LSB}(\text{Key}{\mathbf{i}}) = k{\mathbf{i,0}}$, for each $\mathbf{i} \in \{ \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3} \}$. The navigation path to the leaf corresponding to the key $\text{Key}{\mathbf{0123}}$ is defined as the following string of shuffled key-bits; $$ \begin{aligned} k{\mathbf{0,0}\ } k{\mathbf{1,0}\ } k{\mathbf{2,0}\ } k{\mathbf{3,0}\ } k{\mathbf{0,1}\ } k{\mathbf{1,1}\ } k{\mathbf{2,1}\ } k{\mathbf{3,1}\ } k{\mathbf{0,2}\ } k{\mathbf{1,2}\ } k{\mathbf{2,2}\ } k{\mathbf{3,2}\ } \dots \\ k{\mathbf{0,62}\ } k{\mathbf{1,62}\ } k{\mathbf{2,62}\ } k{\mathbf{3,62}\ } k{\mathbf{0,63}\ } k{\mathbf{1,63}\ } k{\mathbf{2,63}\ } k{\mathbf{3,63} }. \end{aligned} $$ That is, the navigation path to the leaf corresponding to $\text{Key}{\mathbf{0123}}$ is the string of bits composed of: - The least-significant bits of the four 64-bit key parts, $\text{Key}{\mathbf{0}}$, $\text{Key}{\mathbf{1}}$, $\text{Key}{\mathbf{2}}$, $\text{Key}{\mathbf{3}}$, appearing in the order of the key parts as: $$ k{\mathbf{0,0}\ } k{\mathbf{1,0}\ } k{\mathbf{2,0}\ } k{\mathbf{3,0}} $$ - Followed by the second least-significant bits of the four 64-bit key parts, $\text{Key}{\mathbf{0}}$, $\text{Key}{\mathbf{1}}$, $\text{Key}{\mathbf{2}}$, $\text{Key}{\mathbf{3}}$, appearing in the order of the key parts as: $$ k{\mathbf{0,1}\ } k{\mathbf{1,1}\ } k{\mathbf{2,1}\ } k{\mathbf{3,1}} $$ - Then the third least-significant bits of the four key parts, $\text{Key}{\mathbf{0}}$, $\text{Key}{\mathbf{1}}$, $\text{Key}{\mathbf{2}}$, $\text{Key}{\mathbf{3}}$, appearing in the order of the key parts as: $$ k{\mathbf{0,2}\ } k{\mathbf{1,2}\ } k{\mathbf{2,2}\ } k{\mathbf{3,2}} $$ - Up until the most-significant bits of the four key parts, $\text{Key}{\mathbf{0}}$, $\text{Key}{\mathbf{1}}$, $\text{Key}{\mathbf{2}}$, $\text{Key}{\mathbf{3}}$, appearing in the order of the key parts as: $$ k{\mathbf{0,63}\ } k{\mathbf{1,63}\ } k{\mathbf{2,63}\ } k{\mathbf{3,63}} $$!Navigation Path Derivation This construction ensures that in every quadruplet of consecutive path-bits, there is a one-to-one correspondence between the bits and the four parts of the key, $\text{Key}{\mathbf{0}}$, $\text{Key}{\mathbf{1}}$, $\text{Key}{\mathbf{2}}$, $\text{Key}{\mathbf{3}}$. Reconstructing the key from path-bits When executing a basic operation such as an UPDATE of a value at a leaf, one has to reconstruct the key from the remaining key found at the leaf and the path-bits spent in navigating to the leaf. Denote the remaining key as a quadruple, $$ \text{RKey}{\mathbf{0123}} = \big(\text{RKey}{\mathbf{0}} , \text{RKey}{\mathbf{1}} , \text{RKey}{\mathbf{2}} , \text{RKey}{\mathbf{3}}\big) $$ Since the Path was constructed by shuffling key-bits from the four parts, $\text{Key}{\mathbf{0}}$, $\text{Key}{\mathbf{1}}$, $\text{Key}{\mathbf{2}}$, $\text{Key}{\mathbf{3}}$, one would expect the reverse-process (going from the path-bits to the original key) to work just as easily. Perhaps taking the level lvl of a leaf and reducing it modulo 4, should be sufficient to tell which part of the Remaining Key, $\text{RKey}{\mathbf{i}}$, must the Path key-bit be appended to. Example of key reconstruction Suppose the leaf storing the key-value pair $\big(\text{Key}{\mathbf{0123}}, \text{V}{\mathbf{01..7}} \big)$ is reached at level 7, the path-bits used are 0110101, and the remaining key is $$ \text{RKey}{\mathbf{0123}} = \big(\text{RKey}{\mathbf{0}} , \text{RKey}{\mathbf{1}} , \text{RKey}{\mathbf{2}} , \text{RKey}{\mathbf{3}} \big) $$ That is, the path-bits are $$ \begin{aligned} \text{path-bit}6 = 1, \text{ path-bit}5 = 0, \text{ path-bit}4 = 1, \text{ path-bit}3 = 0, \\ \text{ path-bit}2 = 1, \text{ path-bit}1 = 1, \text{ path-bit}0 = 0. \end{aligned} $$ So, in order to place $\text{path-bit}6$, one first computes $7 \text{ modulo } 4$ to get 3. Hence, the $\text{key-bit}6$ must be appended to the third key part, $\text{RKey}{\mathbf{2}}$. Next, one climbs the tree to level 6, where $\text{path-bit}5 = 0$. One then computes $6 \text{ modulo } 4$ and gets 2. The $\text{path-bit}5$ must then be appended to the second key part, $\text{RKey}{\mathbf{1}}$. Again, one climbs the tree to level 5, where $\text{path-bit}4 = 1$. Computing $5 \text{ modulo } 4$ yields 1. The $\text{path-bit}4$ is thence appended to the first key part, $\text{RKey}{\mathbf{0}}$. One then continues in the same fashion: $\text{Climbs the tree to level } 4. \text{ Computes }\ 4 \text{ modulo } 4 = 0. \text{ Appends a path-bit to the fourth part, } \text{RKey}{\mathbf{3}}.$ $\text{Climbs the tree to level } 3. \text{ Computes }\ 3 \text{ modulo } 4 = 3. \text{ Appends a path-bit to the third part, } \text{RKey}{\mathbf{2}}.$ $\text{Climbs the tree to level } 2. \text{ Computes }\ 2 \text{ modulo } 4 = 2. \text{ Appends a path-bit to the second part, } \text{RKey}{\mathbf{1}}.$ $\text{Climbs the tree to level } 1. \text{ Computes }\ 1 \text{ modulo } 4 = 1. \text{ Appends a path-bit to the first part, } \text{RKey}{\mathbf{0}}.$ The next climb is to the root. The navigation path-bits have been exhausted, and the last append has actually completed reconstruction of the key. Leaf levels and integers modulo 4 It is clear, from the above example, that there is a one-to-one correspondence between the integers modulo 4 (i.e., elements of the group $\mathbb{Z}4 = \{ 0, 1, 2, 3 \}$) and remaining key parts $\text{RKey}{\mathbf{0}}$, $\text{RKey}{\mathbf{1}}$, $\text{RKey}{\mathbf{2}}$, $\text{RKey}{\mathbf{3}}$. That is, there is a mapping: $$ 1 \mapsto \text{RKey}{\mathbf{0}},\ \ 2 \mapsto \text{RKey}{\mathbf{1}},\ \ 3 \mapsto \text{RKey}{\mathbf{2}} \text{ and }\ 0 \mapsto \text{RKey}{\mathbf{3}}. $$ The quadruple structure of the path bits and the level of leaves therefore have a homomorphic relationship that can be described in terms of the cyclic group of integers modulo 4, $\mathbb{Z}4 = \{ 0, 1, 2, 3 \}$. Since addition modulo n is an expensive computation in the state machine context, it is important to find a more efficient algorithm to achieve the same result. Alternate cyclic group of order 4 In order to explore cyclic groups of order 4, take the vector $\mathbf{x} = (1,0,0,0)$, and rotate the components of $\mathbf{x}$ one position to the left. Note that, rotating $\mathbf{x} = (1,0,0,0)$ - once, yields $(0,0,0,1)$ - twice, one obtains $(0,0,1,0)$ - thrice, one gets $(0,1,0,0)$ - four times, and the result is $\mathbf{x} = (1,0,0,0)$ Continuously rotating $\mathbf{x} = (1,0,0,0)$ does not result in any other vector but the four vectors $$ \mathbf{G4} = \{ (1,0,0,0),\ (0,0,0,1),\ (0,0,1,0),\ (0,1,0,0) \}. $$ This set of four vectors $\mathbf{G4}$ together with the described rotation, forms a group. In fact, $\mathbf{G4}$ is isomorphic (or homomorphically equivalent) to $\mathbb{Z}4$ under addition modulo 4. That is, there is a natural one-to-one correspondence between the elements of $\mathbb{Z}4$ and those of $\mathbf{G4}$, as follows: $$ 0 \mapsto (1,0,0,0),\ \ 1 \mapsto (0,1,0,0),\ \ 2 \mapsto (0,0,1,0)\ \text{ and }\ 3 \mapsto (0,0,0,1). $$ Note that the four numbers 0, 1, 2 and 3 can be expressed in their binary form with just two bits, and the same one-to-one correspondence holds as: $$ \text{00} \mapsto (1,0,0,0),\ \ \text{01} \mapsto (0,1,0,0),\ \ \text{10} \mapsto (0,0,1,0)\ \text{ and }\ \text{11} \mapsto (0,0,0,1). $$ Special cyclic register for leaf levels Define a register called LEVEL which is a vector of four bits; three 0 bits and one 1 bit. And the operation ROTATELEVEL which is the left rotation of LEVEL's bits by one position. If LEVEL is initialised as $(1,0,0,0)$, observe that applying ROTATELEVEL four times brings LEVEL back to $(1,0,0,0)$. That is, $$ (1,0,0,0) \to (0,0,0,1) \to (0,0,1,0) \to (0,1,0,0) \to (1,0,0,0) $$ Therefore, LEVEL is cyclic under ROTATELEVEL, and is in fact algebraically the same as the cyclic group $\mathbf{G4}$ described above. Using LEVEL register in key reconstruction First note that, when navigating the tree, the leaf level can be indicated by one of the four possible states of the LEVEL register. And this works for all possible leaf levels because, for any positive integer j: $$ \begin{aligned} {\text{LEVEL}} = (1,0,0,0)\ \text{indicates that the leaf level is one of the following};\ 0, 4, 8, \dots , 0 + 4j. \ \\ {\text{LEVEL}} = (0,1,0,0)\ \text{indicates that the leaf level is one of the following};\ 1, 5, 9, \dots , 1 + 4j. \ \\ {\text{LEVEL}} = (0,0,1,0)\ \text{indicates that the leaf level is one of the following};\ 2, 6, 10, \dots, 2 + 4j. \\ {\text{LEVEL}} = (0,0,0,1)\ \text{indicates that the leaf level is one of the following};\ 3, 7, 11, \dots, 3 + 4j. \end{aligned} $$ Second, the two least-significant bits of each of these number, when written in binary, are as follows: $$ \begin{aligned} \text{Each of these numbers};\ 0, 4, 8, \dots , 0 + 4j;\ \text{ends with } 00.\ \ \\ \text{Each of these numbers};\ 1, 5, 9, \dots , 1 + 4j;\ \text{ends with } 01.\ \ \\ \text{Each of these numbers};\ 2, 6, 10, \dots , 2 + 4j; \text{ends with } 10.\ \\ \text{Each of these numbers};\ 3, 7, 11, \dots , 3 + 4j;\ \text{ends with } 11. \end{aligned} $$ It suffices therefore to only read the two least-significant bits of the leaf level in order to determine the position of the bit 1 in the LEVEL register. Third, the position of the bit 1 in the LEVEL register tallies precisely with the part of the remaining key, $\text{RKey}{\mathbf{i}}$, to which the last used path-bit came from. So then, when reconstructing the key, one needs only to check where the bit 1 is in the LEVEL register, because: $$ \begin{aligned} {\text{LEVEL}} = (1,0,0,0)\ \ \text{means, the last used path bit must be appended to } \mathbf{RKey0}.\\ {\text{LEVEL}} = (0,1,0,0)\ \ \text{means, the last used path bit must be appended to } \mathbf{RKey1}.\\ { \text{LEVEL}} = (0,0,1,0)\ \ \text{means, the last used path bit must be appended to } \mathbf{RKey2}.\\ {\text{LEVEL}} = (0,0,0,1)\ \ \text{means, the last used path bit must be appended to } \mathbf{RKey3}. \end{aligned} $$ Since things are rather mechanical in state machines, one or two more functions are needed. For instance, one for initialising the LEVEL register, and another for reading the position of the bit 1. # executor-pil.md: The Storage executor is to the Storage Assembly code a like a slave-worker to the master. It carries out all Storage Actions in accordance with rules and logic the Assembly code has set out. As per instruction of the Main SM, the Storage executor makes function calls to the Storage ROM for a specific secondary Assembly code, stored as a JSON-file, by using the same aforementioned selectors of secondary Assembly codes. For example, if the Main SM requires a new leaf to be created at a found non-zero leaf, the Storage executor uses isSetInsertFound as a function call for the SetInsertFound (or SIF) Storage Action. The Storage executor then proceeds to build committed polynomials and executes the SIF Storage Action. As previously observed, in our very first UPDATE example in this series of documents, all values are expressed as quadruplets of unsigned integers. For example, the Remaining Key looks like this, $$ \texttt{RKey} = \big(\texttt{RKey}0, \texttt{RKey}1, \texttt{RKey}2, \texttt{RKey}3 \big) $$ The executor therefore uses an internal 4-element register called op = [,,,], for handling values from the Storage ROM, which are needed in the internal step-by-step evaluations of the Storage Action being executed. It is thus reset to 0 after every evaluation. All the function calls seen in the Assembly code: $\texttt{GetSibling()}$, $\texttt{GetValueLow()}$, $\texttt{GetValueHigh()}$, $\texttt{GetRKey()}$, $\texttt{GetSiblingRKey()}$, $\texttt{GetSiblingHash()}$, $\texttt{GetSiblingValueLow()}$, $\texttt{GetSiblingValueHigh()}$, $\texttt{GetOldValueLow()}$, $\texttt{GetOldValueHigh()}$, $\texttt{GetLevelBit()}$, $\texttt{GetTopTree()}$, $\texttt{GetTopBranch()}$, and $\texttt{GetNextKeyBit()}$, are actually performed by the Storage executor. The values being fetched are carried with the op register. For instance, if the function call is GetRKey() then the Storage executor gets the RKey from the rom.line file, carries it with op as; $$ \begin{aligned} \texttt{op[0] = ctx.rkey[0]}; \\ \texttt{op[1] = ctx.rkey[1]}; \\ \texttt{op[2] = ctx.rkey[2]}; \\ \texttt{op[3] = ctx.rkey[3]}; \end{aligned} $$ where ctx signifies a Storage Action. Also, since all Storage Actions require some hashing, the Storage SM delegates all hashing actions to the Poseidon SM. However, from within the Storage SM, it is best to treat the Poseidon SM as a blackbox. The Storage executor simply specifies the sets of twelve values to be digested. And the Poseidon SM then returns the required digests of the values. Storage PIL All computations executed in the Storage SM must be verifiable. A special Polynomial identity language (PIL) code is therefore used to set up all the polynomial constraints the verifier needs so as to validate correctness of execution. The preparation for these polynomial constraints actually starts in the Storage executor. In order to accomplish this, the Storage executor uses; selectors, setters and instructions; which are in fact Boolean polynomials. See the list of these Boolean committed polynomials in the table below.*

Selectors	Setters	Instructions
selFree[i]	setHashLeft[i]	iHash
selSiblingValueHash[i]	setHashRight[i]	iHashType
selOldRoot[i]	setOldRoot[i]	iLatchSet
selNewRoot[i]	setNewRoot[i]	iLatchGet
selValueLow[i]	setValueLow[i]	iClimbRkey
selValueHigh[i]	setValueHigh[i]	iClimbSiblingRkey
selRkeyBit[i]	setSiblingValueLow[i]	iClimbSiblngRkeyN
selSiblingRkey[i]	setSiblingValueHigh[i]	iRotateLevel
selRkey[i]	setRkey[i]	iJmpz
	setSiblingRkey[i]	iConst0
	setRkeyBit[i]	iConst1
	setLevel[i]	iConst2
		iConst3
		iAddress

Every time each of these Boolean polynomials are utilised or performed, a record of a "1" is kept in its register. This is called an Execution trace. Therefore, instead of performing some expensive computations in order to verify correctness of execution (at times repeating the same computations being verified), the trace of execution is tested. The verifier takes the execution trace, and tests if it satisfies the polynomial constraints (or identities) in the PIL code. This technique helps the zkProver to achieve succintness as a zero-knowledge proof-verification system. Poseidon hash Poseidon SM is more straightforward once one understands the internal mechanism of the original Poseidon hash function. The hash function's permutation process translates readily to the Poseidon SM states. The Poseidon State Machine carries out Poseidon Actions in accordance with instructions from the Main SM executor and requests from the Storage SM. That is, it computes hashes of messages sent from any of the two SMs, and also checks if the hashes were correctly computed. The zkProver uses Poseidon hash function defined over the Goldilocks field, denoted by $\mathbb{F}p$, where $p = 2^{64} - 2^{32} + 1$. The states of the Poseidon SM coincide with the twelve (12) internal states of the Poseidon^{π} permutation function. These are; in0, in1, ... , in7, hashType, cap1, cap2 and cap3. Poseidon^{π} runs 30 rounds, 3 times. Adding up to a total of 90 rounds. It outputs four (4) hash values; hash0, hash1, hash2 and hash3. !Poseidon HASH0 In the case of the zkProver storage, two slightly different Poseidon hashes are used; HASH0 is used when a branch node is created, whilst HASH1 is used when a leaf node is created. This depends on the hashType, which is a boolean. So Poseidon acts as HASH1 when hashType = 1, and HASH0 when hashType = 0. Since the Poseidon hash outputs $4 \lfloor(63.99)\rfloor \text{ bits} = 252$, and one bit is needed to encode each direction. The tree can therefore have a maximum of 252 levels. # index.md: A standard state machine is characterized by sets of states (as inputs) stored in registers, instructions on how the states should transition, and the resultant states (as outputs) stored as new values in the same registers. The below figure demonstrates a standard state machine. !A Generic State Machine Some state machines can be monolithic, serving as prototypes for a specific computation, while others may specialize in performing multiple computations of the same type. Depending on the computational algorithm, a state machine may have to run through a number of state transitions before producing the desired output. Iterations of the same sequence of operations may be required, to the extend that most common state machines are cyclic by nature. The Storage state machine is one of the secondary zkProver state machines responsible for all operations on data stored in the zkProver's storage. It receives instructions from the Main state machine, called Storage Actions. The Main state machine performs typical database operations such as: Create, Read, Update and Delete (CRUD); and then instructs the Storage state machine to verify whether these were correctly performed. A microprocessor-type state machine The Storage SM is in fact a micro-processor with both the firmware and the hardware parts. It is in the firmware part of the Storage SM where the logic and rules are set up, expressed in JSON format and stored in a ROM. The zero-knowledge Assembly (zkASM) is a language developed by the team and especially designed to map instructions from the zkProver Main SM to other state machines, in this case, to the Storage SM's executor. The Main SM's instructions, or Storage Actions, are passed to the Storage SM executor for execution in compliance with the rules and logic specified in the JSON-file. The hardware part uses another novel language, called Polynomial Identity Language (PIL), which is also developed by the team and especially designed for the zkProver, because almost all state machines express computations in terms of polynomials. State transitions in state machines must satisfy computation-specific polynomial identities. In order for the Storage SM to carry out Storage Actions, its executor generates committed and constant polynomials, which are then checked against polynomial identities to prove that computations were correctly executed. See the Design approach section for how this achieved. zkProver's storage As a means to achieve zero-knowledge, all data is stored in the form of Merkle trees. This means the Storage SM often makes requests of another state machine, the Poseidon SM, to perform hashing (referred to as Poseidon Actions). The Main SM performs computations on the key-value data stored in special Merkle trees, called Sparse Merkle trees (SMTs). The keys and values are stored as strings of 256 bits and can be interpreted as 256-bit unsigned integers. The mechanics of the Storage SM and its basic operations are described in detail in later sections of this documentation. They cover: 1. The basic design of the zkProver's Storage and some preliminaries. Also explains how the Sparse Merkle trees (SMTs) are built. 2. Explanations of the basic operations routinely performed on these SMTs. 3. Details about specific parameters the Storage SM uses, such as how keys and paths are created, and the two Poseidon hash functions used in the SMTs. 4. The three main components of the Storage SM: - The Storage SM Assembly - The Storage SM executor, and - The Storage SM PIL code, for all the polynomial identities and proving correctness of execution. # mechanism.md: The Storage SM is one of zkProver's secondary state machines, and thus receives instructions from the Main SM called Storage Actions. The Storage Actions basically refer to the verification of CRUD operations executed by the Main SM. The Storage SM is designed as a microprocessor and it is therefore composed of three parts: - Storage Assembly code, - Storage executor code, - Storage PIL code. Storage assembly The Storage Assembly is the interpreter between the Main State Machine and its own Executor. The Storage SM receives instructions from the Main SM written in zkASM. It then generates a JSON-file containing the corresponding rules and logic, which are stored in a special ROM for the Storage SM. The Storage SM has a primary Storage Assembly code that maps each instruction of the Main SM to the secondary Assembly code corresponding to each basic operation. These basic operations are mainly the CREATE, READ, UPDATE and DELETE, as discussed in previous sections. Considering some special cases, there are eight secondary Storage Assembly codes all-in-all, each for a distinct basic operation. We list these in the table below. | Storage Actions | File Names | Code Names | Action Selectors In Primary zkASM Code | | --------------------------------- | ------------------ | ---------- | ------------------------------------ | | READ | Get | Get | isGet() | | UPDATE | SetUpdate | SU | isSetUpdate() CREATE new value at a found leaf | SetInsertFound | SIF | isSetInsertFound() | | CREATE new value at a zero node | SetInsertNotFound | SINF | isSetInsertNotFound() | | DELETE last non-zero node | SetDeleteLast | SDL | isSetDeleteLast() | | DELETE leaf with non-zero sibling | SetDeleteFound | SDF | isSetDeleteFound() | | DELETE leaf with zero sibling | SetDeleteNotFound | SDNF | isSetDeleteNotFound() | | SET a zero node to zero | SetZeroToZero | SZTZ | isSetZeroToZero() | Input and output states of the Storage SM are literally SMTs, given in the form of; the Merkle roots, the relevant siblings, as well as the key-value pairs. Note that state machines use registers in the place of variables. All values needed, for carrying out the basic operations, are stored by the primary Assembly code in the following registers; HASHLEFT, HASHRIGHT, OLDROOT, NEWROOT, VALUELOW, VALUEHIGH, SIBLINGVALUEHASH, RKEY, SIBLINGRKEY, RKEYBIT, LEVEL. The SIBLINGVALUEHASH and SIBLINGRKEY registers are only used by the SetInsertFound and the SetDeleteFound secondary Assembly codes. The rest of the registers are used in all the secondary Assembly codes. SMT Action selectors in primary assembly code The Primary Assembly Code maps the Main SM instructions to the relevant Storage Actions using selectors. Like switches can either be ON or OFF, selectors can either be 1 or 0, where 1 means the action is selected for execution, while 0 means the instruction does not tally with the required action so a "jump if zero" JMPZ is applied. The primary Assembly code uses selectors by following the sequence in which these Storage Actions are listed in the above table. That is, - It first checks if the required action is a Get. If it is so, the storagesmget.zkasm code is fetched for execution. - If not, it checks if the required action is SetUpdate. If it is so, the storagesmsetupdate.zkasm code is fetched for execution. - If not, it continues to check if the required action is SetInsertFound. If it is so, the storagesmsetinsertfound.zkasm code is fetched for execution. - If not, it continues in the same way until the correct action is selected, in which case the corresponding code is fetched for execution. That's all the primary Storage Assembly code does and the details of how each of the SMT Actions is stipulated in the individual secondary Assembly codes. The primary and secondary Storage Assembly files are stored as JSON-files in the Storage ROM, ready to be fetched as function calls by the Storage Executor. The UPDATE zkASM code Take as an example the SetUPDATE zkASM code. The primary Storage Assembly code uses the selector isSetUpdate() for SetUPDATE. Note that an UPDATE involves the following actions: 1. Reconstructs the corresponding key, from both the remaining key found at the leaf and key-bits used to navigate to the leaf. 2. Ascertains that indeed the old value was included in the old root, 3. Carries out the UPDATE of the old value with the new value, as well as updating all nodes along the path from the leaf to the root. There is only one SetUPDATE Assembly code, storagesmsetupdate.zkasm, for all the above three computations. Key reconstruction in zkASM Key Reconstruction is achieved in two steps: positioning of the bit "1" in the LEVEL register, and using the LEVEL register to climb the RKey. That is, append the path bit last used in navigation to the correct RKey part. 1. Positioning the bit "1" in the LEVEL register The SetUPDATE zkASM code, first initialises the LEVEL register to (1,0,0,0). Then uses the GetLevelBit() function to read the two least-significant bits of the leaf level, which happens in two cases, each with its own two subcases: Case 1. If the least-significant bit of leaf level is 0, then the GetLevelBit() function is used again to read the second least-significant bit of the leaf level. - Subcase 1.1: If the second least-significant bit of the leaf level is 0, it means the leaf level is a multiple of 4, which is equivalent to 0 because leaf level works in modulo 4. So, the LEVEL register must remain as (1,0,0,0). - Subcase 1.2: If the second least-significant bit of the leaf level is 1, it means the leaf level in its binary form ends with a 10. Hence, leaf level is a number of the form 2 + 4k, for some positive integer k. As a result, the LEVEL register must be rotated to the position, (0,0,1,0). The code therefore applies ROTATELEVEL twice to LEVEL = (1,0,0,0) in order to bring it to (0,0,1,0). Case 2. If the least-significant bit of leaf level is 1; then, the LEVEL register is rotated three times to the left, using ROTATELEVEL, and bringing the LEVEL register to (0,1,0,0). Next, the GetLevelBit() function is used again to read the second least-significant bit of the leaf level. - Subcase 2.1: If the second least-significant bit of the leaf level is 0, it means the leaf level in its binary form ends with a 01. That is, leaf level is a number of the form 1 + 4k, for some positive integer k. And thus, the LEVEL register must remain in its current position, (0,1,0,0). So it does not need to be rotated. - Subcase 2.2: Otherwise, the second least-significant bit of the leaf level is 1, which means the leaf level in its binary form ends with a 11. Hence, leaf level is a number of the form 3 + 4k, for some positive integer k. Consequently, the LEVEL register needs to be rotated from the current position (0,1,0,0) to the position (0,0,0,1). 2. Using LEVEL to "climb the RKey" The Remaining Key is fetched using the GetRKey() function and stored in the RKEY register. When climbing the tree, there are two functions that are used in the code: the CLIMBRKEY and the ROTATELEVEL. - First, the LEVEL register is used to pinpoint the correct part of the Remaining Key to which the path-bit last used in the navigation must be appended. - Second, the ROTATELEVEL is used to rotate the LEVEL register once. - The CLIMBRKEY is used. Firstly, to shift the value of the pinpointed RKey part one position to the left. Secondly, to insert the last used path bit to the least-significant position of the shifted-value of the pinpointed RKey part. The above two steps are repeated until all the path bits used in navigation have been appended. Later, equality between the reconstructed key and the original key is checked. Checking inclusion of old value in old root The above key reconstruction, together with checking inclusion of the old value in the old root and updating the old value to the new value, are carried out simultaneously. Since checking inclusion of the old value in the old root follows the same steps as the update of the old value to the new value, the corresponding lines in the Assembly code are similar. It suffices therefore to explain only one of these two computations. Next is the discussion of the update of the old value to the new value. Update part of SetUPDATE All values, $\text{V}{0123}=\big(\text{V}{0},\text{V}{1},\text{V}{2},\text{V}{3},\text{V}{4},\text{V}{5},\text{V}{6},\text{V}{7}\big)$ are 256-bit long and expressed as lower half and higher half as, VALUELOW $=\big(\text{V}{0},\text{V}{1},\text{V}{2},\text{V}{3}\big)$ and VALUEHIGH $=\big(\text{V}{4},\text{V}{5},\text{V}{6},\text{V}{7} \big)$. 1. Computing the new leaf value 1. The functions GetValueLow() and GetValueHigh() are used to fetch VALUELOW $=\big(\text{V}{0},\text{V}{1},\text{V}{2},\text{V}{3}\big)$ and VALUEHIGH $=\big(\text{V}{4},\text{V}{5},\text{V}{6},\text{V}{7}\big)$, respectively. 2. The VALUELOW $= \big(\text{V}{0},\text{V}{1},\text{V}{2},\text{V}{3}\big)$ is stored in a register called HASHLEFT, whilst VALUEHIGH $=\big(\text{V}{4},\text{V}{5},\text{V}{6},\text{V}{7}\big)$ is stored in another register called HASHRIGHT. 3. The hashed value of $\text{V}{0123}$ is computed using HASH0 as, $\text{HASH0}\big(\text{HASH}\ {\text{LEFT}}\|\text{HASH}\ {\text{RIGHT}}\big)$. Note that this is in fact, $\text{POSEIDON}\big(0\|0\|0\|0\|\text{VALUE}\ {\text{LOW}}\|\text{VALUE}\ {\text{HIGH}}\big)$. The hashed value is then stored in HASHRIGHT. !!!info This means the HASHRIGHT and the HASHLOW are 'make-shift' registers. Whenever a value is stored in it, the old value that was previously stored therein is simply pushed out. They hold values only for the next computation. 4. Next the Rkey is copied into the HASHLEFT register. And the leaf value is computed by using HASH1 as, $\text{HASH1}\big(\text{HASH}\ {\text{LEFT}}\|\text{HASH}\ {\text{RIGHT}}\big)$. i.e., The value of the leaf is, $\text{HASH1}\big(\text{RKey}\|\text{HashedValue}\big)$. The leaf value is then copied into another register called NEWROOT. 2. Climbing the SMT Check if the path bit that led to the leaf is 0 or 1, by using the GetNextKeyBit() function. Case 1: If the path bit (called 'key bit' in the code) is 0, then the corresponding sibling is on the right. Therefore, using 'jump if zero' JMPZ, the code jumps to the SUSiblingIsRight routine. - The leaf value in NEWROOT is pushed into the HASHLEFT register. - The hash value of the sibling node is fetched, using the GetSiblingHash() function. And it is pushed into the HASHRIGHT register. - The hash value of the parent node is computed using HASH0 as follows, $\text{HASH0}\big(\text{HASH}\ {\text{LEFT}} \|\text{HASH}\ {\text{RIGHT}}\big)$. The parent node is $\text{POSEIDON}\big(0\|0\|0\|0\|\text{LeafValue}\|\text{SiblingHash}\big)$. Case 2: If the path bit is 1, then the corresponding sibling is on the left. The routine SUSiblingIsRight is then executed. - The leaf value in NEWROOT is pushed into the HASHRIGHT register. - The hash value of the sibling node is fetched, using the GetSiblingHash() function. And it is pushed into the HASHLEFT register. - The hash value of the parent node is computed using HASH0 as follows, $\text{HASH0}\big(\text{HASH}\ {\text{LEFT}}\|\text{HASH}\ \text{RIGHT}\big)$. The parent node is $\text{POSEIDON}\big(0\|0\|0\|0\|\text{SiblingHash}\|\text{LeafValue}\big)$. 3. Check if tree top has been reached The code uses the function GetTopTree() to check is the top of the tree has been reached. Case 1. If GetTopTree() returns 1, then Step 2 is repeated. But this time using the hash value of the corresponding sibling at the next level (at leaf level - 1). Case 2. If GetTopTree() returns 0, then the code jumps to the SULatch routine. The SULatch is an overall routine for the entire SetUPDATE Assembly code. It is here where, - Equality between the reconstructed key and the original key is checked. - Equality between the computed old root value and the original old root is checked. Once consistency is established both between the keys and the old roots, then all new values; the new root, the new hash value, and the new leaf value are set using LATCHSET. Remaining secondary assembly codes The Assembly codes for the other seven SMT Actions to a certain extent follow a similar pattern except for a few cases where especially adjusted routines are used. Actions such as: 1. The SetInsertFound (or SIF) may involve a change in the topology of the SMT by extending a branch once or several times. In cases where a branch has been extended, the SIF Assembly code, when computing the new root, uses another routine called SIFClimbBranch just for updating values along the newly extended branch. This is done in addition to the SIFClimbTree, which is the exact same routine as the aforementioned SUClimbTree of the SetUPDATE case. It is for the same reason that SIF Assembly utilizes special registers: the SIBLINGVALUEHASH and SIBLINGRKEY. 2. The opposite SMT Action, the SetDeleteFound or SDF, may entail a previously extended branch being reversed. As in the SIF case, if a branch had been extended but now the extension needs to be reversed due to a deleted leaf value, a special routine called SDFClimbBranch is used when updating values of nodes along the newly shortened branch. This SDFClimbBranch routine is the exact same routine as the SIFClimbBranch. Similarly, the SDF Assembly code uses the SDFClimbTree as in the SetUPDATE Assembly. Note also that there is only one Get Assembly code for the READ SMT Action, and the rest of the secondary Assembly codes are Set Assembly codes differing according to their respective SMT Actions. So Get uses LATCHGET at the end of a run, while the Set codes use LATCHSET. # circom-intro-brief.md: !!!info In this document, we describe the CIRCOM component of the zkProver. It is one of the four main components of the zkProver, as outlined here. These principal components are; the Executor or Main SM, STARK Recursion, CIRCOM, and Rapid SNARK. You may refer to the original CIRCOM paper for more details. As seen in the zkProver overview document, the output of the STARK recursion component is a STARK proof. The next step in the zkProver's process of providing validity proof is to produce the witness similar to the output of the STARK Recursion. Although the zkProver is designed as a state machine emulating the EVM, in order to generate this witness, it makes use of a tool based on the Arithmetic circuits model, called CIRCOM. CIRCOM takes the output of the STARK Recursion as input, so as to create its corresponding witness. The witness is in turn taken as an input to the Rapid SNARK, which is used to generate a SNARK proof published as the validity proof. !CIRCOM's input and output In fact, CIRCOM takes a STARK proof as input and produces its corresponding Arithmetic circuit, expressed as the equivalent set of equations called rank-1 constraint system (R1CS). The set of valid circuit input, intermediate and output values satisfying the R1CS is actually the witness related to the input STARK proof. This document focuses on what CIRCOM is, its common implementation context, and how the zkProver utilizes CIRCOM. Circuit context Arithmetic circuits are mostly used as standard models for studying the complexity of computations. An Arithmetic circuit is composed of addition and multiplication gates, and wires that carry values that are elements of a prime finite field $\mathbb{F}p$, where p is typically a very large prime number. In the context of ZK-proof protocols, a prover can use an Arithmetic circuit to prove knowledge of a valid assignment to all wires of the circuit. And if the proof is correct, the verifier is convinced that the computation expressed as the Arithmetic circuit is valid, but learns nothing about the wiresâ€™ assigned values. Arithmetic circuits are commonly encoded in the form of R1CS. Once obtained, the R1CS can later be used by a zk-SNARK protocol to generate verifiable proof. A valid proof attests to the fact that the prover knows an assignment to all wires of the circuit that fulfills all the constraints of the R1CS. An issue that arises, when applying zero-knowledge protocols to complex computations such as a circuit describing the logic of a ZK-rollup, is that the number of constraints to be verified can be extremely large. CIRCOM was developed for the very purpose of scaling complex Arithmetic circuits by realizing them as combined instantiations of smaller Arithmetic circuits. What is CIRCOM? CIRCOM is a Domain-Specific Language (DSL) used to define Arithmetic circuits, and it has an associated compiler of Arithmetic circuits to their respective R1CS. !CIRCOM Overall Context CIRCOM as a DSL As described in the title of its specifications paper, CIRCOM is a language for building complex zero-knowledge circuits. It is designed as a low-level circuit language, mimicking the design of electronic circuits, for naturally defining Arithmetic circuits. As a DSL, it allows programmers to design and create Arithmetic circuits of their own choice, and later on apply these circuits to ZK tools. One of the main peculiarities of CIRCOM is its modularity as a language. It allows the definition of parameterizable small circuits called templates, which can be instantiated to form part of larger circuits. In this regard, CIRCOM users can use templates to create their own custom circuits with varied complexity. However, CIRCOM users can also use templates from CIRCOMLIB, which is a publicly available library that contains hundreds of circuits such as; comparators, hash functions, digital signatures, binary and decimal converters. Circuit compiler In addition to being a DSL used to define and create Arithmetic circuits, CIRCOM has a special compiler of Arithmetic circuits into their equivalent R1CS. The term "CIRCOM" is in fact short for Circuit Compiler, implying that the CIRCOM compiler takes an Arithmetic circuit as input and outputs the following: - A file with the R1CS description, containing the set of associated R1CS constraints, and - A program written either in C++ or WebAssembly, for efficiently computing a valid assignment to all wires of the circuit (the witness). The CIRCOM compiler is mainly written in Rust and it is open source. Features The CIRCOM language has its own peculiarities. The focus in this subsection is on a few features characteristic of CIRCOM. Consider as an example, the Multiplier circuit with input signals $\texttt{a}$ and $\texttt{b}$, and an output signal $\texttt{c}$ satisfying the constraint $\texttt{a} \times \texttt{b} \texttt{ - c = 0}$. !A simple Multiplier Arithmetic circuit The figure above depicts a simple Multiplier Arithmetic circuit with input wires labeled $\texttt{a}$ and $\texttt{b}$ and an output wire labeled $\texttt{c}$ such that $\mathtt{a} \times \mathtt{b = }\ \texttt{c}$. The wires are referred to as signals. The constraint related to this Multiplier circuit is: $$ \mathtt{a} \times \mathtt{b} \mathtt{\ -\ c = 0} $$ The pragma instruction The pragma instruction specifies the version of the CIRCOM compiler being used. It is meant to ensure compatibility between the circuit and the compiler version. If the two are incompatible, the compiler throws a warning. pragma circom 2.0.0; As a precautionary measure, all files with the .circom extension should start with a pragma instruction. In the absence of this instruction, it is assumed that the code is compatible with the latest compiler version. Declaration of signals In the Multiplier example, there are two input signals $\texttt{a}$ and $\texttt{b}$, and an output signal $\texttt{c}$. In CIRCOM, each signal is given an identifier. However, the same sympols; $\texttt{a}$, $\texttt{b}$ and $\texttt{c}$ used for signals can also be identifiers. The three signals are declared as follows, signal input a; signal input b; signal output c; The <== operator The functionality of this operator is twofold; - On the one hand, it sets a constraint that expresses that the value of $\texttt{c}$ must be the result of multiplying

$\texttt{a}$ by $\texttt{b}$, - On the other hand, the operator instructs the compiler in how to generate the program that computes the assignment of circuit signals. c <== a b; Creation of templates One of the main peculiarities of CIRCOM is the allowance to define parameterizable small circuits called templates. Templates are parametrizable in the sense that their outputs depend on free input values (i.e., values that are freely chosen by users). They are general descriptions of circuits, that have some input and output signals, as well as a relation between the inputs and the outputs. The following code shows how a Multiplier template is created: pragma circom 2.0.0; template Multiplier () { // declaration of signals signal input a; signal input b; signal output c; // constraints c <== a b; } Instantiation of templates Although the above code succeeds in creating the Multiplier template, the template is yet to be instantiated. In CIRCOM, the instantiation of a template is called a component, and it is created as follows: component main = Multiplier(); The Multiplier template does not depend on any parameter. However, it is possible to initially create a generic, parametrizable template that can later be instantiated using specific parameters to construct the circuit. Declaration of components is the means by which CIRCOM enables programmers to work in a modular fashion. Small circuits can be defined which can be combined to create larger circuits by the complexity of the computations needed to be carried out. Compiling a circuit As previously mentioned, the use of the operator "<==" in the Multiplier template has dual functionality: - It captures the arithmetic relation between signals. - It also provides a way to compute $\texttt{c}$ from $\texttt{a}$ and $\texttt{b}$. In general, the description of a CIRCOM circuit also keeps dual functionality. That is, it performs both symbolic tasks and computational tasks. This enables the compiler to easily generate the R1CS describing a circuit, together with instructions to compute intermediate and output values of a circuit. Given a circuit with the multiplier.circom extension, the following line of code instructs the compiler to carry out the two types of tasks: circom multiplier.circom --r1cs --c --wasm --sym After compiling the .circom circuit, the compiler returns four files, - A file with the R1CS constraints (symbolic task). - A C++ program for computing values of the circuit wires (computational task). - A WebAssembly program for computing values of the circuit wires (computational task). - A file of symbols for debugging and printing the constraint system in an annotated way (symbolic task). At this stage, either one of the C++ or WebAssembly programs generated by the compiler can be used to compute all signals that match the set of constraints of the circuit. Whichever program is used, needs as input a file containing a set of valid input values. Recall that a valid set of circuit input, intermediate and output values is called the witness. Private and public signals Depending on the template being used, some signals are private while others are public. In the case of the Multiplier template, a signal is private by default, unless it is declared to be public in the instantiation of the template as shown below. component main {public [a]} = Multiplier(); According to the above line, the input signal $\texttt{a}$ is public, while $\texttt{b}$ is private by default. Main component The CIRCOM compiler needs a specific component as an entry point. And this initial component is called main. In the same way the Multiplier template needed instantiation as a component, so does the main component. However, unlike other intermediate components, the main component defines the global input and output signals of a circuit. Denote a list of $\texttt{n}$ signals by $\{\mathtt{ s1, s2, \dots , sn }\}$. The general syntax to specify the main component is the following: component main {public [s1,..,sn]} = templateID(v1,..,vn); Specifying the list of public signals of the circuit, indicated as $\mathtt{public [s1,..,sn]}$, is optional. Note that global inputs are considered private signals while global outputs are considered public. However, the main component has a special attribute to set a list of global inputs as public signals. The rule of thumb is: Any other input signal not included in this list $\mathtt{public [s1,..,sn]}$, is considered private. Concluding CIRCOM's features There are many more features that distinguish CIRCOM from other known ZK tools. The rest of these features delineated in the original CIRCOM paper. We conclude this document by putting together the mentioned CIRCOM features in one code example. The Multiplier template is again used as an example. But the pragma instruction is omitted for simplicity's sake. template Multiplier() { signal input a; signal input b; signal output c; c <== a b; } component main {public [a]} = Multiplier(); # evm-basics.md: In this document we dig deeper into the main state machine or executor component of the zkProver. It is one of the four main components of the zkProver, outlined here. These are - Executor, STARK recursion, CIRCOM, and Rapid SNARK. Since the design of the zkProver emulates that of the EVM, this document focuses on explaining the basics of EVM. Overview of Polygon zkEVM Polygon zkEVM is an L2 network that implements a special instance of the EVM. It emulates the EVM in that the zkProver, which is core to proving and verifying computation correctness, is also designed as a state machine or a cluster of state machines. The terms state machine and virtual machine are used interchangeably in this documentation. Although the Polygon zkEVM architecture and state differ from Ethereum, communication with the Polygon zkEVM is done through a JSON-RPC interface, which is fully compatible with Ethereum RPC. This allows all Ethereum-compatible applications and tools to be natively compatible with the Polygon zkEVM. Since Polygon zkEVM is a separate instance, with a state distinct from Ethereum, balances in accounts may differ and therefore L1 smart contracts cannot be directly accessed through L2 transactions. A special zkEVM bridge and cross-chain messaging mechanism have been developed so as to enable the exchange of data between both networks. Basics of EVM The Ethereum blockchain is a distributed digital ledger that keeps track of all transactions and interactions that occur on the Ethereum network. In addition to recording transactions, the EVM can store and execute smart contracts. These smart contracts are low-level codes that can perform a variety of tasks and operations on the network. The EVM is therefore the computational engine of the Ethereum blockchain responsible for smart contract deployment and execution. The EVM is categorized as a quasiâ€"Turing-complete state machine because it can handle all execution processes except that, due to the gas limit set in every smart contract, is limited to computations with a finite number of steps. At any given point in time, the current state of the Ethereum blockchain is defined by a collection of the blockchain data. An Ethereum state therefore includes account balances, smart contract code, smart contract storage, and other information relevant to the operation of the network. Since Ethereum is a distributed digital ledger, its state is maintained by each of the network's full-nodes. Key features of EVM In terms of how it operates, the EVM is described as; deterministic, sandboxed, and stack-based. Deterministic: For any given input, it always produces the exact same output. This feature is critical for ensuring dependability and predictability of smart contract execution, as well as enabling reliable verification of execution. Sandboxed: Transactions processed by smart contracts run in an environment that is isolated from the rest of the system, making it impossible for transactions to access or modify data outside this environment. This contributes towards network security by preventing unauthorized access to sensitive data. Stack-based: It employs a last-in-first-out (LIFO) memory data structure for processing operations, with data being pushed onto a stack and popped off as needed. Components of EVM The EVM is made up of several components that work together to execute smart contracts on the Ethereum blockchain and provide the above-mentioned features. !EVM Components Involved in the Processing of a Transaction The main components of the EVM involved in the processing of a transaction are: 1. Smart contract bytecode, which is the low-level code executed by the EVM. Each bytecode is a sequence of opcodes (machine-level instructions). And each opcode in an EVM bytecode corresponds to a specific operation, such as arithmetic, conditional branching or memory manipulation. The EVM executes bytecode in a step-by-step fashion, with each opcode being processed in a given sequence. In general, smart contracts are written in a high-level programming language, such as Solidity or Vyper, and then compiled into an EVM bytecode. 2. Processing environment refers to the component responsible for executing smart contracts. It provides a runtime environment for the smart contract bytecode to execute in and manage the memory and storage used by smart contract. 3. Stack is the LIFO data structure used to execute the EVM's operations, and thus turning the EVM into a stack-based machine. 4. Memory is the component that allows smart contracts to store and retrieve data. It is organized as a linear array of bytes, while data is accessed by specifying its location in memory. 5. calldata refers to the set of parameters & values required for a smart contract to perform its function. The transaction that invokes a particular smart contract must contain the right calldata, and thus pass the calldata to that smart contract. calldata is read-only and therefore smart contracts cannot modify it during execution. The smart contract's input data is part of the transaction which is stored on the blockchain, and therefore any changes to the input data would result in a different transaction-hash and hence a different state of the blockchain. 6. Storage is the EVMâ€™s storage component where smart contracts can also store data. It is a persistent key-value store that is associated with each smart contract, and it can be used to store state information. The EVM is a variant of the Von Neumann architecture which means it uses a single shared memory for both data and instructions. - The smart contractâ€™s bytecode is stored in memory in the EVM, and the program counter (PC) keeps track of the current instruction being executed. - Stack is used for storing small values, such as integers and booleans, values needed for immediate use, such as function parameters, local variables, and return values. - Memory is used for storing large data structures, such as arrays and strings. EVM computational costs The EVM has its own instruction set or list of available opcodes, which is a set of low-level commands used to manipulate data in the stack, memory and storage components. The instruction set includes operations such as Arithmetic, bit manipulation and control flow. Additionally, in order to prevent spam and denial of service (DoS) attacks, the EVM employs a gas system. Gas is a unit of measurement for the computational resources required to execute a smart contract, and each operation in the instruction set has its own gas cost. Stack The EVM is a stack-based machine because it uses a stack data structure to execute its operations. When an operation is performed, values that are currently top of the stack are popped off, used in the executed operation, and then the result of the operation is pushed back onto the stack. Some of the main stack operations executed in the EVM are: 1. PUSH: This opcode pushes a value onto the stack. It is usually followed by: - A byte which indicates the number of bytes to be pushed onto the stack, and - The actual bytes to be pushed onto the stack. For example, the opcode PUSH2 0x0123 pushes the bytes 0x01 and 0x23 onto the stack as one word 0x0123. 2. POP: Removes the top value from the stack and discards it. 3. DUP: Duplicates the top value on the stack and pushes the duplicate onto the stack. 4. SWAP: Swaps the top two values on the stack. 5. ADD, SUB, MUL, DIV, MOD: These opcodes perform specific Arithmetic operations on the top two values of the stack, and push the result back onto the stack. 6. AND, OR, XOR, NOT: These opcodes perform bitwise logic operations on the top two values of the stack, and push the result back onto the stack. 7. EQ, LT, GT: These opcodes perform comparison operations on the top two values of the stack, and push the result back onto the stack as a Boolean. 8. SHA3: Computes the SHA3 hash of the top value on the stack, and pushes the hash onto the stack. 9. JUMP, JUMPI: These opcodes modify the program counter, allowing the program to jump to a different part of the code. The EVM stack is limited to 1024 elements. This yields the capacity of (1024×256) bits because each EVM word is 256 bits long. If a contract attempts to PUSH more elements onto the stack, exceeding the 1024-limit, a stack overflow error occurs, causing the transaction to fail. Memory The EVM Memory is used for storing large data structures, such as arrays and strings. It is a linear array of bytes used by smart contracts to store and retrieve data. The size of the memory is dynamically allocated at runtime, meaning that the amount of memory available to a smart contract can grow depending on its needs. EVM Memory is byte-addressable, which means that each byte in the memory can be individually addressed using a unique index. EVM word size is 256 bits (or 32 bytes), which means data is typically loaded and stored in 32-byte chunks. The EVM also provides instructions for loading and storing smaller chunks of data, such as individual bytes or 16-bit words. EVM Memory is referred to as non-persistent or volatile, because the data it stores gets cleared as soon as the execution of a smart contract is completed. This means the EVM needs to have a special component, called the EVM Storage, for permanently storing results of smart contract execution. Itâ€™s also worth noting that, since accessing and modifying EVM Memory consumes computational resources, which are paid for in the form of gas, its use is subject to gas costs. Managing EVM memory - When a contract calls another contract, a new execution environment is created with its own memory space. - The parent contractâ€™s memory space is saved, and the new contractâ€™s memory space is initialized. The new contract can then make use of its memory as needed. - When the called contractâ€™s execution is completed, the memory space is released and the parent contractâ€™s saved memory is restored. It is worth noting that if a smart contract does not actually use the memory it has been allocated, that memory cannot be reclaimed or reused in the execution context of another contract. The opcodes related to memory are as follows: - MLOAD is an opcode used to load a 32-byte word from Memory into the stack. It takes a Memory address as its input and pushes the value stored at that address onto the stack. - MSTORE is an opcode used to store a 32-byte word from the stack into Memory. It takes a Memory address and a value from the stack as its input, and stores the value at the specified address. - MSTORE8 is an opcode similar to MSTORE, except that it stores a single byte of data instead of a 32-byte word. It takes a Memory address and a byte value from the stack as its input, and stores the byte at the specified address. - MSIZE is an opcode that returns the size of the current Memory area in bytes. Storage EVM storage is a persistent key-value storage associated with each smart contract. It is organized as a large array of 32-byte words and each word is identified by a unique 256-bit key, which is used to access and modify the value stored in that word. Since the EVM Storage is non-volatile, data stored in it persists even after the smart contract execution has been completed. Accessing and modifying storage is a relatively expensive operation in terms of gas costs. EVM storage is implemented as a modified version of the Merkle Patricia Tree data structure, which allows for efficient access and modification of the storage data. A Patricia Tree is a specific type of a trie designed to be more space-efficient than a standard trie, by storing only the unique parts of the keys in the tree. Patricia Trees are particularly useful in scenarios where keys share common prefixes, as they allow for more efficient use of memory and faster lookups compared to standard tries. The following opcodes are used to manipulate the storage of a smart contract: - SLOAD loads a 256-bit word from Storage at a given index, and pushes it onto the stack. - SSTORE stores a 256-bit word to Storage at a given index. The value to be stored is popped from the stack, and its index is specified as the next value on the stack. Transaction process Processing transactions in the EVM involves, - the Recursive Length Prefix (RLP) encoding and decoding of transaction data, - the verification of signatures, - the execution of transactions, - storing output values. RLP decoding Transaction data are encoded for storage and decoded for processing. The Recursive Length Prefix (RLP) encoding and decoding is used for this purpose. The first step in processing an Ethereum transaction is to therefore decode the transaction. Transactions are decoded so as to obtain relevant information such as; the recipient's address, the amount of ETH being transferred, and the data payload. Signature verification Every transaction is digitally signed with a signature, which is generated using the Elliptic Curve Digital Signature Algorithm (ECDSA). The ECDSA signature is represented by three (3) values, generally denoted as $\texttt{r}, \texttt{s}, \texttt{v}$. Since the signature, or in particular the triplet $(\texttt{r}, \texttt{s}, \texttt{v})$, was computed from the secret key which is uniquely associated with the address of the Ethereum account (being debited), the three values $(\texttt{r}, \texttt{s}, \texttt{v})$ are sufficient to accurately verify that the transaction has been signed by the owner of the Ethereum account. The Ethereum account is identified by a 20-byte (160-bit) address. The address is derived from the public key associated with the Ethereum account. It is in fact the last 20 bytes of the 256-bit Keccak hash of the public key. Processing a transaction The EVM begins by creating a context with an empty stack and memory space. The bytecode instructions are then executed. The execution involves values being pushed and popped onto and from the stack as required. EVM uses a program counter to keep track of which instruction to execute next. Each opcode has a fixed number of bytes, so the program counter increments by the appropriate number of bytes after each instruction is executed. Stack elements and word size Stack elements are 32 bytes in size. This means each value pushed onto the stack by an opcode, as well as each value popped off the stack by an opcode, are each 32 bytes in size. The 32-byte size limit for stack elements is a fundamental design choice in Ethereum, and is purely based on the size of the EVM word. The EVM word is the basic unit of storage and processing in the EVM, defined as a 256-bit (32-byte) unsigned integer. Since the EVM word is the smallest unit of data that can be processed by the EVM, the stack elements are conveniently set to be of the same size. Summary The EVM sequentially executes the opcodes in the bytecode, by following the program counter. It also manipulates 32-byte values on the stack and in Memory, in accordance with computations required to be performed, and permanently stores relevant values as required. EVM interpreter EVM interpreter is a software component that processes and executes Ethereum transactions. Ethereum smart contracts are written in a high-level programming language such as Solidity, Vyper, Fe or Yul, but are compiled into bytecodes. A bytecode is a set of instructions which the EVM nterpreter processes and executes. Each bytecode is a sequence of Ethereum opcodes. The Ethereum opcodes are low-level instructions set for the EVM, and represent the basic operations that the EVM can perform during the execution of a smart contract triggered by a transaction. A complete list of Ethereum opcodes includes over 200 different operations, ranging from general Arithmetic and Logical operations to more advanced and blockchain environment-specific operations, like calls to other contracts, contract creation, and storage management. Some of the most commonly used opcodes include: - ADD, SUB, MUL, DIV, which are basic arithmetic operations. - CALL, DELEGATECALL, CALLCODE, which are calls to other contracts. - PUSH, POP, which are stack management operations. - JUMP, JUMPI, which respectively refer to jumps and conditional jumps, from one line of a program to another. - SLOAD, SSTORE, which are Storage management operations. - MLOAD, MSTORE, which are Memory management operations. References The Ethereum yellow paper entails technical details on the Ethereum with some of the opcodes listed and described on Pages 30 to 38. A more elaborate exposition of the Ethereum Blockchain is provided in the book Mastering Ethereum by Andreas M. Antonopoulos and Gavin Wood. The shortest and less technical paper on Ethereum is Beigepaper: An Ethereum Technical Specification by Micah Dameron. # index.md: This section covers some of the basic concepts crucial to understanding the design approach of Polygon zkEVM. Since Polygon zkEVM emulates the EVM, a few EVM basics are herein detailed. One of the differences between Polygon zkEVM and Ethereum is in the way their states are recorded. Ethereum uses Patricia Merkle tries while Polygon zkEVM uses Sparse Merkle trees (SMTs). The Concepts section therefore discusses how SMTs are constructed and the operations executable on the SMTs. These are Create, Read, Update and Delete, or simply CRUD. The design approach is delineated in terms of an example: the multiplicative Fibonacci state machine. Further details of the Polygon zkEVM's state machine design are given in the form of a 'Generic state machine', which involves a program written in zkASM called the ROM. This section also includes a brief discussion on what CIRCOM is, and how it is used in the zkProver. # ending-program.md: Dealing with negative numbers Before looking at ways to properly end a program, let us first make a few remarks on how to handle negative numbers. Recall that all values of the execution trace belong to a field $\mathbb{F}p$ where $p=2^{64} â^'2^{32} +1$. Since $\mathtt{(p-x) + x \equiv p \texttt{ modulo } p}$ for all $\mathtt{x \in \mathbb{F}p }$ and $\mathtt{p \equiv 0 \texttt{ modulo } p}$, it follows that $\mathtt{-x \equiv p-x \texttt{ modulo } p}$. Any negative number $\mathtt{-a}$ is therefore interpreted as $\mathtt{p â^' a}$. Consider the same zkASM program with four instructions, as discussed before, except that instead of moving a positive constant 3 into registry $\mathtt{B}$, it now moves a negative constant $\mathtt{-3}$. The execution trace, where the free input is still $\mathtt{7}$, is as a follows. $$ \begin{aligned} \begin{array}{|l|c|c|c|c|c|c|c|c|}\hline \texttt{ }\texttt{ }\texttt{ }\texttt{ }\texttt{ }\texttt{ } \bf{Instructions } & \texttt{FREE} & \texttt{CONST}& \texttt{setB}& \mathtt{setA}& \texttt{inFREE}& \mathtt{inB} & \mathtt{inA} & \mathtt{A} & \mathtt{A'} & \mathtt{B} & \mathtt{B'} \\ \hline \mathtt{\$\{getAFreeInput()\} => A} & \texttt{7} & \texttt{0} & \texttt{0} & \texttt{1} & \texttt{1} & \texttt{0} & \texttt{0} & \mathtt{0} & \mathtt{7} & \mathtt{0} & \mathtt{0} \\ \hline \texttt{ } \mathtt{-3 => B} & \texttt{0} & \texttt{p-3} & \texttt{1} & \texttt{0} & \texttt{0} & \texttt{0} & \texttt{0} & \mathtt{7} & \mathtt{7} & \mathtt{0} & \mathtt{p-3} \\ \hline \texttt{ } \mathtt{:ADD } & \texttt{0} & \texttt{0} & \texttt{0} & \texttt{1} & \texttt{0} & \texttt{1} & \texttt{1} & \mathtt{7} & \mathtt{4} & \mathtt{p-3} & \mathtt{p-3} \\ \hline \texttt{ } \mathtt{:END } & \texttt{0} & \texttt{0} & \texttt{1} & \texttt{1} & \texttt{0} & \texttt{0} & \texttt{0} & \mathtt{4} & \mathtt{0} & \mathtt{p-3} & \mathtt{0} \\ \hline \end{array} \end{aligned} $$ Using conditional jumps The FFTs are most efficient for polynomials of degree $\mathtt{T + 1 = 2^N}$. This causes descrepancies when the program being executed has fewer instructions than an appropriate power of 2. Consider the following program, with only five instructions, and its corresponding execution trace. $$ \begin{aligned} \begin{array}{|l|c|} \hline \texttt{ }\texttt{ }\texttt{ }\texttt{ }\texttt{ }\texttt{ } \bf{Instructions } & \texttt{FREE} & \texttt{CONST}& \texttt{setB} & \mathtt{setA}& \texttt{inFREE}& \mathtt{inB} & \mathtt{inA} & \mathtt{A} & \mathtt{A'} & \mathtt{B} & \mathtt{B'} \\ \hline \texttt{ } \mathtt{\$\{getAFreeInput()\} => A} & \texttt{7} & \texttt{0} & \texttt{0} & \texttt{1} & \texttt{1} & \texttt{0} & \texttt{0} & \mathtt{0} & \mathtt{7} & \mathtt{0} & \mathtt{0} \\\hline \texttt{ } \mathtt{3 => B} \qquad\qquad\qquad\qquad\quad & \texttt{0} & \texttt{3} & \texttt{1} & \texttt{0} & \texttt{0} & \texttt{0} & \texttt{0} & \mathtt{7} & \mathtt{7} & \mathtt{0} & \mathtt{3} \\\hline \texttt{ } \mathtt{:ADD } \qquad\qquad\qquad\quad\quad\quad\text{ }\text{ } & \texttt{0} & \texttt{0} & \texttt{0} & \texttt{1} & \texttt{0} & \texttt{1} & \texttt{1} & \mathtt{7} & \mathtt{10} & \mathtt{3} & \mathtt{3} \\\hline \texttt{ }

\mathtt{:END } \qquad\qquad\qquad\quad\qquad\text{}\text{ }\text{ } & \texttt{0} & \texttt{0} & \texttt{0} & \texttt{1} & \texttt{0} & \texttt{1} & \texttt{1} & \mathtt{10} & \mathtt{13} & \mathtt{3} & \mathtt{3} \\\hline \texttt{ } \mathtt{:END } \qquad\qquad\qquad\quad\qquad\text{}\text{ }\text{ } & \texttt{0} & \texttt{0} & \texttt{1} & \texttt{1} & \texttt{0} & \texttt{0} & \texttt{0} & \mathtt{13} & \mathtt{0} & \mathtt{3} & \mathtt{0} \\\hline \end{array} \end{aligned} $$ The question is: How (or when) to end the program when the trace has size $\mathtt{5}$ and the polynomials have are of degree $\mathtt{2^3 = 8}$? NaÃ¯ve way to end a program A simple and naÃ¯ve fix could be to repeating the "$\texttt{:END}$" instruction in order to fill the $\mathtt{8 - 5}$ rows of the execution trace. However, for the sake of preserving the cyclicity of the state machine, the last instruction still has to ensure that the next values of registries must revert back to their initial values. $$ \begin{aligned} \begin{array}{|l|c|}\hline \texttt{ }\texttt{ }\texttt{ }\texttt{ }\texttt{ } \bf{Instructions } & \texttt{FREE} & \texttt{CONST}& \texttt{setB}& \texttt{setA}& \texttt{inFREE}& \mathtt{inB} & \mathtt{inA} & \mathtt{A} & \mathtt{A'} & \mathtt{B} & \mathtt{B'} \\ \hline \mathtt{\$\{getAFreeInput()\} => A} & \texttt{7} & \texttt{0} & \texttt{0} & \texttt{1} & \texttt{1} & \texttt{0} & \texttt{0} & \mathtt{0} & \mathtt{7} & \mathtt{0} & \mathtt{0} \\\hline \mathtt{3 => B} \qquad\qquad\qquad\qquad\quad & \texttt{0} & \texttt{3} & \texttt{1} & \texttt{0} & \texttt{0} & \texttt{0} & \texttt{0} & \mathtt{7} & \mathtt{7} & \mathtt{0} & \mathtt{3} \\\hline \mathtt{:ADD } \qquad\qquad\qquad\quad\quad\text{ }\text{ } & \texttt{0} & \texttt{0} & \texttt{0} & \texttt{1} & \texttt{0} & \texttt{1} & \texttt{1} & \mathtt{7} & \mathtt{10} & \mathtt{3} & \mathtt{3} \\\hline \mathtt{:END } \qquad\qquad\qquad\quad\qquad\text{}\text{ }\text{ } & \texttt{0} & \texttt{0} & \texttt{0} & \texttt{1} & \texttt{0} & \texttt{1} & \texttt{1} & \mathtt{10} & \mathtt{13} & \mathtt{3} & \mathtt{3} \\\hline \mathtt{:END } \qquad\qquad\qquad\quad\qquad\text{}\text{ }\text{ } & \texttt{0} & \texttt{0} & \texttt{1} & \texttt{1} & \texttt{0} & \texttt{0} & \texttt{0} & \mathtt{13} & \mathtt{0} & \mathtt{3} & \mathtt{0} \\\hline \mathtt{:END } \qquad\qquad\qquad\quad\qquad\text{}\text{ }\text{ } & \texttt{0} & \texttt{0} & \texttt{1} & \texttt{1} & \texttt{0} & \texttt{0} & \texttt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} \\\hline \mathtt{:END } \qquad\qquad\qquad\quad\qquad\text{}\text{ }\text{ } & \texttt{0} & \texttt{0} & \texttt{1} & \texttt{1} & \texttt{0} & \texttt{0} & \texttt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} \\\hline \mathtt{:END } \qquad\qquad\qquad\quad\qquad\text{}\text{ }\text{ } & \texttt{0} & \texttt{0} & \texttt{1} & \texttt{1} & \texttt{0} & \texttt{0} & \texttt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} \\\hline \end{array} \end{aligned} $$ Note that, since there are no restrictions placed on the next values of $\mathtt{A'}$ and $\mathtt{B'}$, we can set these to any values. In the above case, both $\mathtt{A'}$ and $\mathtt{B'}$ are set to zeros. Programs with conditional jumps We now add to the zkASM program, the instruction $$ \texttt{jump } \texttt{to } \texttt{a } \texttt{particular } \texttt{address } \texttt{if } ... $$ denoted by $\mathtt{JMPZ(addr)}$. This means the executor must jump to the specified position in the program on condition that the preceding operation, denoted by $\texttt{op}$, is zero. Since this instruction needs to specify the destination of the jump, we need to add the line column so as to indicate which row is each instruction placed in the program. See the table below. $$ \begin{aligned} \begin{array}{|l|c|}\hline \texttt{ line } & \bf{Instructions } \text{ }\text{ }\text{ }\text{ } \\ \hline \quad\texttt{ 0 } & \text{ }\mathtt{\$\{getAFreeInput()\} => A} \text{ }\\ \hline \quad\texttt{ 1 } & \text{ }\mathtt{-3 => B} \qquad\qquad\qquad\quad \\ \hline \quad\texttt{ 2 } & \text{ }\mathtt{:ADD } \qquad\qquad\quad\quad\quad\text{ }\text{ } \\ \hline \quad\texttt{ 3 } & \text{ }\mathtt{A : JMPZ(5) } \quad\qquad\quad\quad\text{ }\text{ } \\ \hline \quad\texttt{ 4 } & \text{ }\mathtt{:ADD } \qquad\qquad\quad\quad\quad\text{ }\text{ } \\ \hline \quad\texttt{ 5 } & \text{ }\mathtt{:END } \qquad\qquad\quad\quad\qquad\text{}\text{ }\text{ } \\ \hline \end{array} \end{aligned} $$ With regards to the instruction "$\texttt{A:JMPZ(5)}$" in $\texttt{line}$ $\texttt{3}$ of the zkASM program, 1. The $\texttt{A}$ registry, preceding the colon, means that $\texttt{op}$ is set to the value of $\texttt{A}$. 2. If $\texttt{op = 0}$, the program jumps to $\texttt{line}$ $\texttt{5}$. 3. If $\mathtt{op \not= 0}$, the program continues sequentially with the instruction at $\texttt{line}$ $\texttt{4}$. Conditional jumps examples Suppose the executor of our SM reads a zkASM program where one of its instructions requires a free input to be taken. The aim here is to observe how a variation in free inputs affects the length of the execution trace. For the sake of brevity, we present execution traces with only five columns (i.e., we omit the columns corresponding to the selectors and setters). Example A In this example, the free input is $\mathtt{FREE = 7}$. Focusing on $\texttt{line}$ $\texttt{3}$: Since in the previous operation, $\mathtt{op=ADD}$ and $\mathtt{A = A + B = 4 \not= 0}$, and thus the condition for the $\texttt{JMPZ}$ is not satisfied. There must be no jump. The execution continues sequentially to the instruction in $\texttt{line}$ $\texttt{4}$. The resulting execution trace is therefore 6 steps long. $$ \begin{aligned} \begin{array}{|l|c|c|c|c|c|c|c|}\hline \texttt{ line } & \bf{Instructions }\text{ }\text{ }\text{ }\text{ } & \mathtt{FREE} & \mathtt{A} & \mathtt{A'} & \mathtt{B} & \mathtt{B'} \\\hline \quad\texttt{ 0 } & \text{ }\mathtt{\$\{getAFreeInput()\} => A} \text{ } & \quad\mathtt{7} & \mathtt{0} & \mathtt{7} & \mathtt{0} & \mathtt{0\ }\\\hline \quad\texttt{ 1 } & \text{ }\mathtt{-3 => B} \qquad\qquad\qquad\quad & \quad\mathtt{0} &\mathtt{7} & \mathtt{7} & \mathtt{0} & \mathtt{-3\ }\\\hline \quad\texttt{ 2 } & \text{ }\mathtt{:ADD } \qquad\qquad\quad\quad\quad\text{ }\text{ } & \quad\mathtt{0} & \mathtt{7} & \mathtt{4} & \mathtt{-3\ } & \mathtt{-3\ }\\\hline \quad\texttt{ 3 } & \text{ }\mathtt{A : JMPZ(5) } \quad\qquad\quad\quad\text{ }\text{ } & \quad\mathtt{0} & \mathtt{4} & \mathtt{4} & \mathtt{-3\ } & \mathtt{-3\ }\\\hline \quad\texttt{ 4 } & \text{ }\mathtt{:ADD } \qquad\qquad\quad\quad\quad\text{ }\text{ } & \quad\mathtt{0} & \mathtt{4} & \mathtt{1} & \mathtt{-3\ } & \mathtt{-3\ }\\\hline \quad\texttt{ 5 } & \text{ }\mathtt{:END } \qquad\qquad\quad\quad\qquad\text{}\text{ }\text{ } & \quad\mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{-3\ } & \mathtt{0}\\\hline \end{array} \end{aligned} $$ Example B In this case, the free input is $\mathtt{FREE = 3}$. Again, focusing on $\texttt{line}$ $\texttt{3}$: Since in the previous operation, $\mathtt{op=ADD}$ and $\mathtt{A = A + B = 0}$, it means the condition for the $\texttt{JMPZ}$ is satisfied. Hence the executor jumps to $\texttt{line}$ $\texttt{5}$ as per instruction in $\texttt{line}$ $\texttt{3}$. The execution trace is now only 5 steps long. $$ \begin{aligned} \begin{array}{|l|c|c|c|c|c|c|c|}\hline \texttt{ line } & \texttt{Instructions } \text{ }\text{ }\text{ }\text{ } & \mathtt{FREE} & \mathtt{A} & \mathtt{A'} & \mathtt{B} & \mathtt{B'} \\\hline \quad\texttt{ 0 } & \text{ }\mathtt{\$\{getAFreeInput()\} => A} \text{ } & \quad\mathtt{3} & \mathtt{0} & \mathtt{3} & \mathtt{0} & \mathtt{0\ }\\\hline \quad\texttt{ 1 } & \text{ }\mathtt{-3 => B} \qquad\qquad\qquad\quad & \quad\mathtt{0} &\mathtt{3} & \mathtt{3} & \mathtt{0} & \mathtt{-3\ }\\\hline \quad\texttt{ 2 } & \text{ }\mathtt{:ADD } \qquad\qquad\quad\quad\quad\text{ }\text{ } & \quad\mathtt{0} & \mathtt{3} & \mathtt{0} & \mathtt{-3\ } & \mathtt{-3\ }\\\hline \quad\texttt{ 3 } & \text{ }\mathtt{A : JMPZ(5) } \quad\qquad\quad\quad\text{ }\text{ } & \quad\mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{-3\ } & \mathtt{-3\ }\\\hline \quad\texttt{ 5 } & \text{ }\mathtt{:END } \qquad\qquad\quad\quad\qquad\text{}\text{ }\text{ } & \quad\mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{-3\ } & \mathtt{0}\\\hline \end{array} \end{aligned} $$ We observe that the inclusion of conditional jumps in the zkASM programs introduces dynamics to the executor's output. For example, the instructions in the zkASM program (in Example B) are not sequentially executed. This requires more agility in our proving system, more especially that our aim is to lighten the verifier's work. Correctness checks for state machines with jumps The dynamism brought about by the inclusion of jumps in our zkASM programs means more checks need to be added so that the state machine can properly prove correct behaviour. Since the instructions require the executor to perform varied operations, and due to the presence of jumps, these operations are not sequentially executed, we then need to do a few more checks; 1. Check program operations. Every operation being executed needs to be checked if it is the correct one. That is, if the instruction is an $\texttt{ADD}$, then we must check that indeed an $\texttt{ADD}$ was performed and not a different operation. 2. Check instructionsâ€™ sequence. The sequence in which the operations are executed must tally with the instructions in the zkASM program, and not necessarily their usual chronological sequence of the lines of code. e.g., The lines of instructions executed in Example B above, are lines $\texttt{0, 1, 2, 3, 5}$, where $\texttt{line}$ $\texttt{4}$ was skipped. Note that more complicated programs can be built where, for instance, the SM execution jumps to a previous line of code, and thus repeating execution of some instructions. 3. Check correct program ending. How the program ends also needs to be managed. Due to the presence of jumps, the length of the execution trace is no longer constant for the same program if the free inputs are varied. 4. Check positioning of Publics. We must ensure that all $\texttt{publics}$ (the inputs and the outputs) are in the expected positions. So, $\texttt{publics}$ should be placed at known steps. This ensures that the SM's PIL does not have to change with every execution. For this reason, $\texttt{publics}$ are going to be placed in either the first positions of the polynomial or the last positions of the polynomial (these are specified positions in the arrays representing the columns of the trace). 5. Check correct Program (ROM) Execution. Although the polynomial identities are there to monitor correct state transitions (checking that each instruction does what it is supposed to do), and also that the correct sequence of instructions is followed, one still needs to make sure the instruction being executed belongs to the program in the first place. So, if for instance, the third instruction is being executed, is it really the third instruction of the right program or not. One of the implications of a dynamic execution trace, due to the inclusion of jumps, is that some of the previously constant (or preprocessed) polynomials must now be committed polynomials. If we are going to allow polynomials (i.e., columns in the execution trace) corresponding to the instructions to be committed, we need to be cautious that only instructions belonging to the right program are being executed. For this purpose, we use a tool called Plookup. # exec-trace-correct.md: In this document we discuss how the correctness of the execution trace is ensured. The first step is to build a mechanism for verifying correctness of the execution trace. This requires creating a set of arithmetic constraints that only hold true when the execution trace is correct. These arithmetic constraints are equations that registry values in any two consecutive rows of the correct execution trace, must satisfy. Similar to the mFibonacci SM, where each state had to conform to polynomial identities, the arithmetic constraints of the generic state machine are translated into polynomial identities and ultimately into the PIL language. Constructing arithmetic constraints Since these arithmetic constraints govern state transitions, they express the next state $\big(\mathtt{A'} ,\mathtt{B'}\big)$ in terms of the current state $\big(\texttt{A},\texttt{B}\big)$. That is, in terms of the execution trace, the new registry values are linear combinations of previous registry values, together with constants and free inputs. We therefore need auxiliary columns for selectors. Like switches that can either be ON or OFF, selectors too can either have a value $\mathtt{1}$ or $\mathtt{0}$, depending on the instruction being executed. In continuing with the previous example of a four-instruction state machine, - We use selectors; $\texttt{inFREE}$, $\texttt{inA}$, $\texttt{inB}$, $\texttt{setA}$ and $\texttt{setB}$; corresponding to the columns; $\texttt{FREE}$, $\texttt{A}$, $\texttt{B}$, $\mathtt{A'}$ and $\mathtt{B'}$, respectively. - If the instruction being executed involves a column $\texttt{X}$, then the corresponding selector $\texttt{inX}$ must have the value $\mathtt{1}$, otherwise $\texttt{inX}$ must be $\mathtt{0}$. - If the instruction being executed moves the result of the computation into column $\mathtt{X'}$, then the corresponding selector $\texttt{setX}$ must have the value $\mathtt{1}$, otherwise $\texttt{setX}$ must be $\mathtt{0}$. - Notice that $\texttt{CONST}$ does not need a selector. If a computation constant is not needed in an instruction, the corresponding value in the $\texttt{CONST}$ column is simply set to $\texttt{0}$. The arithmetic constraints are therefore defined by the following linear combinations; $$ \mathtt{Aâ€² = A + setA \cdot \big(inA \cdot A + inB \cdot B + inFREE \cdot FREE + CONST - A \big)}\ \ \tag{Eqn 1a} $$ $$ \mathtt{Bâ€² = B + setB \cdot \big(inA \cdot A + inB \cdot B + inFREE \cdot FREE + CONST - B \big)}\ \ \tag{Eqn 1b} $$ The figure below depicts the linear combinations of our state machine as an algebraic processor of sorts. !The Generic State Machine as an Algebraic Processor The vertical gray box (with the "+" sign) in the above figure denotes addition. It expresses forming linear combinations of some of the columns; $\texttt{FREE}$, $\texttt{A}$, $\texttt{B}$, or $\texttt{CONST}$. Each is either included or excluded from the linear combination depending on whether their corresponding selectors have the value $\mathtt{1}$ or $\mathtt{0}$. An extra register denoted by $\texttt{op}$ acts as a carrier of intermediate of the computation being executed and waiting to be placed in the correct output register (on the right in above figure), depending on the values of $\texttt{setA}$ and $\texttt{setB}$. Testing arithmetic constraints We now test if the arithmetic constraints tally with each of the four instructions of our program. 1. The first instruction: "$\mathtt{\$\{getAFreeInput()\} => A}$" The first instruction involves a free input 7 and this free input is moved into registry $\texttt{A}$, as its the next value. Therefore, by definition of the selectors, $\mathtt{inFREE = 1}$ and $\mathtt{setA = 1}$. Also, the value of the other selectors is $\texttt{0}$. Substituting these values in the above arithmetic constraints yields; $$ \mathtt{Aâ€² = A + 1 \cdot \big(0 \cdot A + 0 \cdot B + 1 \cdot 7 + 0 - A \big) = A + (7 - A) = 7}\text{ } $$ $$ \mathtt{Bâ€² = B + 0 \cdot \big(0 \cdot A + 0 \cdot B + 1 \cdot 7 + 0 - B \big) = B}\qquad\qquad\qquad \\ $$ This illustrates that the value of the free input was moved into $\texttt{A}$, while $\texttt{B}$ remains unaltered. Hence, the first instruction was correctly executed. 2. The second instruction: "$\mathtt{3 => B}$" The second instruction involves the $\mathtt{CONST}$ column, and the constant value $\texttt{3}$ is moved into registry $\texttt{B}$, as its next value. Consequently, $\mathtt{CONST = 3}$ and $\mathtt{setB = 1}$. All other selectors have the value $\texttt{0}$. Again, substituting these values in the arithmetic constraints yields; $$ \mathtt{Aâ€² = A + 0 \cdot \big(0 \cdot A + 0 \cdot B + 0 \cdot FREE + 3 - A \big) = A}\qquad\qquad\qquad $$ $$ \mathtt{Bâ€² = B + 1 \cdot \big(0 \cdot A + 0 \cdot B + 0 \cdot FREE + 3 - B \big) = B + (3 - B) = 3} $$ This shows that the value of $\texttt{A}$ was not changed, but the constant value $\mathtt{3}$ was moved into $\texttt{B}$. And thus, the second instruction was correctly executed. 3. The third instruction, "$\mathtt{:ADD }$" This instruction involves the registries $\texttt{A}$ and $\texttt{B}$, and the result is moved into registry $\texttt{A}$, as its the next value. This means, the values of the corresponding selectors are as follows; $\mathtt{inA = 1}$, $\mathtt{inB = 1}$ and $\mathtt{setA = 1}$. The arithmetic constraints become; $$ \mathtt{Aâ€² = A + 1 \cdot \big(1 \cdot A + 1 \cdot B + 0 \cdot FREE + 0 - A \big) = A + (A + B - A) = A + B}\text{ } $$ $$ \mathtt{Bâ€² = B + 0 \cdot \big(1 \cdot A + 1 \cdot B + 0 \cdot FREE + 0 - B \big) = B}\qquad\qquad\qquad\qquad\quad $$ The sum of the registry values in $\mathtt{A}$ and $\mathtt{B}$ was moved into $\texttt{A}$, while $\texttt{B}$ remains unmodified, proving that the third instruction was correctly executed. 4. The fourth instruction, "$\mathtt{:END }$" The fourth instruction moves the initial registry values (i.e., $\mathtt{A = 0}$ and $\mathtt{B0 = 0}$) into registries $\texttt{A}$ and $\texttt{B}$, as their next values, respectively. As a result, values of the corresponding selectors are; $\mathtt{setA = 1}$ and $\mathtt{setB = 1}$. Substitutions into the arithmetic constraints give us the following; $$ \mathtt{Aâ€² = A + 1 \cdot \big(0 \cdot A + 0 \cdot B + 0 \cdot FREE + 0 - A \big) = A - A = 0} $$ $$ \mathtt{Bâ€² = B + 1 \cdot \big(0 \cdot A + 0 \cdot B + 0 \cdot FREE + 0 - B \big) = B - B = 0} $$ Clearly, the next registry values of both $\mathtt{A}$ and $\mathtt{B}$ are reset to zeros as per the fourth instruction. The execution trace can now be updated to reflect the selector columns, as shown below. $$ \small \begin{aligned} \begin{array}{|l|c|c|c|c|c|c|c|}\hline \texttt{ }\texttt{ }\texttt{ }\texttt{ }\texttt{ }\texttt{ } \bf{Instructions } & \texttt{FREE} & \texttt{CONST}& \texttt{setB}& \mathtt{setA}& \texttt{inFREE}& \mathtt{inB} & \mathtt{inA} & \mathtt{A} & \mathtt{A'} & \mathtt{B} & \mathtt{B'} \\ \hline \texttt{ } \mathtt{\$\{getAFreeInput()\} => A} & \texttt{7} & \texttt{0} & \texttt{0} & \texttt{1} & \texttt{1} & \texttt{0} & \texttt{0} & \mathtt{0} & \mathtt{7} & \mathtt{0} & \mathtt{0} \\ \hline \texttt{ } \mathtt{3 => B} \qquad\qquad\qquad\qquad\quad & \texttt{0} & \texttt{3} & \texttt{1} & \texttt{0} & \texttt{0} & \texttt{0} & \texttt{0} & \mathtt{7} & \mathtt{7} & \mathtt{0} & \mathtt{3} \\ \hline \texttt{ } \mathtt{:ADD } \qquad\qquad\qquad\quad\quad\quad\text{ }\text{ } & \texttt{0} & \texttt{0} & \texttt{0} & \texttt{1} & \texttt{0} & \texttt{1} & \texttt{1} & \mathtt{7} & \mathtt{10} & \mathtt{3} & \mathtt{3} \\ \hline \texttt{ } \mathtt{:END } \qquad\qquad\qquad\quad\qquad\text{}\text{ }\text{ } & \texttt{0} & \texttt{0} & \texttt{1} & \texttt{1} & \texttt{0} & \texttt{0} & \texttt{0} & \mathtt{10} & \mathtt{0} & \mathtt{3} & \mathtt{0} \\ \hline \end{array} \end{aligned} $$ Remarks 1. The $\texttt{CONST}$ column stores the constants of the computation. It should however, not be mistaken for a constant polynomial. The term 'constant' refers to the fact that the column contains constants of the computations. 2. It shall be seen later, in our implementation of a state machine with jumps, that $\texttt{CONST}$ is in fact a committed polynomial rather than a constant polynomial. 3. All operations in the constraints are carried out $\mathtt{\ modulo }$ the order p of the prime field. The so-called Goldilocks-like field, with $p = 2^{64} â^' 2^{32} +1$, is mainly used where 64-bit numbers suffice (see Plonky2). Otherwise, the BN128 field is used. In order to match the type of commitment scheme used in the zkEVM, these arithmetic constraints must first be expressed as polynomial identities, which are in turn compiled with PILCOM. Deeper context for the executor Up to this stage, we have only mentioned that the SM executor reads instructions in a program written in zkASM, and it may take some free inputs in order to produce an execution trace. There is however a lot more detail that goes into this process. For example, the executor does not really read the zkASM program as is. But rather, the zkASM program (which we name here, program.zkasm), is first compiled with a tool called zkasmcom, into a JSON file (call the JSON file, program.json). Also, the free inputs may come in the form of another JSON file, let's name it input.json. In addition, the executor can read information in databases and receive relevant input such as the PIL.json. See below diagram for a concise description of what the executor does. !Figure 5: SM executor in a broader context Although the execution trace is composed of the evaluations of the committed polynomials and the evaluations of the constant polynomials, the two evaluations do not happen simultaneously. Instead, the constant polynomials are preprocessed only once, because they do not change and are specific for a particular state machine. The committed polynomials, on the other hand, can vary. And are therefore only processed as and when their corresponding verifiable proof is required. Polynomial identities For verification purposes, the execution trace needs to be interpolated into polynomial identities. The general theory on interpolation of polynomials involves technical terms such as; Lagrange's interpolation, roots of unity and Fast Fourier Transforms. However, these concepts are not prerequisites because the way we construct the execution trace makes sure the polynomial identities follow readily from the arithmetic constraints. In the above example of a zkASM program with four instructions, we can use the fourth-roots of unity $\mathcal{H} = \{ 1 = \omega^4, \omega, \omega^2, \omega^3 \} \subset \mathbb{F}p$ to interpolate, such that the columns, $$ \mathtt{ A, inA, setA, B, inB, setB, FREE, inFREE, CONST } $$ correspond to polynomials in $\texttt{x}$, where $\mathtt{x = \omega^i}$, written (without changing symbols) as, $$ \mathtt{ A(x), inA(x), setA(x), B(x), inB(x), setB(x), FREE(x), inFREE(x), CONST(x)}. $$ That is, according to the execution trace in Table 3 above, these polynomials are equivalent to the column-arrays, as follows, $$ \begin{aligned} &\mathtt{ A = [0,7,7,10]}\text{ }\text{ } \iff\text{ } \ \mathtt{A(x) = A(\omega^i) = A[i]} \\ &\mathtt{ B = [0,0,3,3]} \text{ }\text{ } \iff\text{ } \ \mathtt{B(x) = B(\omega^i) = B[i] } \\ &\mathtt{ inA = [0,0,1,1]\text{ }\text{ } \iff\text{ } \ \mathtt{inA(x) = inA(\omega^i) = inA[i]} } \\ &\mathtt{ inB = [0,0,1,0]} \text{ }\text{ } \iff\text{ } \ \mathtt{inB(x) = inB(\omega^i) = inB[i]} \\ &\mathtt{ setA = [1,0,1,1] \text{ }\text{ } \iff\text{ } \ \mathtt{setA(x) = setA(\omega^i) = setA[i] } } \\ &\mathtt{ setB = [0,1,0,1]} \text{ }\text{ } \iff\text{ } \ \mathtt{setB(x) = setB(\omega^i) = setB[i]} \\ &\mathtt{ FREE = [7,0,0,0]}\text{ }\text{ } \iff\text{ } \ \mathtt{FREE(x) = FREE(\omega^i) = FREE[i]} \\ &\mathtt{ CONST = [1,0,0,0]}\text{ }\text{ } \iff\text{ } \ \mathtt{CONST(x) = CONST(\omega^i) = CONST[i]} \\ &\mathtt{ inFREE = [1,0,0,0]} \text{ }\text{ } \iff\text{ } \ \mathtt{inFREE(x) = inFREE(\omega^i) = inFREE[i]} \\ \end{aligned} $$ The arithmetic constraints seen above as $\mathtt{Eqn 1}$, are easily written as polynomial identities, as follows, $$ \begin{aligned} \mathtt{A(x\omega) - \big(A(x) + setA(x) \cdot \big(op(x) - A(x) \big) \big) = 0} \\ \mathtt{B(x\omega) - \big(B(x) + setB(x) \cdot \big(op(x) - B(x) \big) \big) = 0} \end{aligned} $$ where $\mathtt{op(x) = inA(x) \cdot A(x) + inB(x) \cdot B(x) + inFREE(x) \cdot FREE(x) + CONST(x)}$. As far as boundary constraints are concerned, we can, for instance: - Make both the free input and the last registry value of $\mathtt{A}$ public. - Create public 'variables', $\texttt{input}$ and $\texttt{output}$. - Set boundary constraints, $$ \begin{aligned} &\mathtt{L1(x) \cdot \big(FREE(\omega^0) - input\big) = 0} \\ &\mathtt{L2(x) \cdot \big(A(\omega^{3}) - output\big) = 0} \end{aligned} $$ where $\mathtt{L1(x)}$ and $\mathtt{L2(x)}$ are precomputed constant polynomials. In fact, $\mathtt{L1(x) = [1,0,0,0]}$ and $\mathtt{L2(x) = [0,0,0,1]}$. In the big scheme of things, these are Lagrange polynomials emanating from interpolation. Verification relies on the fact that: these polynomial identities, including the boundary constraints, hold true if, and only if the execution trace is correct and faithful to the instructions in the zkASM program. The PIL description of the SM executor, reading instructions from the zkASM program with four instructions, is depicted in the figure provided below. !The PIL description for the 4-instruction program # intro-generic-sm.md: In this document is an introduction of the basic components of a generic state machine. Unlike the mFibonacci state machine, which is an implementation of one specific computation, we now describe a generic state machine that can be instantiated with various computations of the user's choice. The idea here is to create a state machine that behaves like a processor of sorts. In as much as a processor has registries and a clock, so is our generic state machine. It receives instructions in the form of programs written in Assembly, and makes state transitions at each clock in accordance with these instructions. See the figure below, for a state machine with registries $\texttt{A}$ and $\texttt{B}$, and a state $\big(\texttt{A}^{\texttt{i}},\texttt{B}^{\texttt{i}}\big)$ that changes to another state $\big(\texttt{A}^{\texttt{i+2}},\texttt{B}^{\texttt{i+2}}\big)$ in accordance with two instructions, $\texttt{Instruction}{\texttt{i}}$ and $\texttt{Instruction}{\texttt{i+1}}$. !Figure 1: A typical generic state machine The aim with this document is to explain how the machinery used in the mFibonacci SM; to execute computations, produce proofs of correctness of execution, and verify these proofs; can extend to a generic state machine. Think of our state machine as being composed of two parts; the part that has to do with generating the execution trace, while the other part is focused on verifying that the executions were correctly executed. - The former part is more like the 'software' of the state machine, as it is concerned with interpreting program instructions and correctly generating the execution trace. A novel language dubbed the zero-knowledge Assembly (zkASM) language is used in this part. - But the latter part is more like the 'hardware' as it consists of a set of arithmetic constraints (or their equivalent, polynomial identities) that every correctly generated execution trace must satisfy. Since these arithmetic constraints are transformed into polynomial identities (via an interpolation process), they are described in a novel language called the Polynomial Identity Language (PIL). Generic SM executor As seen with the mFibonacci SM, the SM executor takes certain inputs together with the description of the SM, in order to produce the execution trace specifically corresponding to these inputs. !Figure 2: mFibonacci state machine producing input-specific execution trace The main difference, in the Generic state machine case, is the inclusion of a program which stipulates computations to be carried out by the SM executor. These computations could range from a simple addition of two registry values, or moving the value in registry $\texttt{A}$ to registry $\texttt{B}$, to computing some linear combination of several registry values. !Figure 3: A Generic state machine producing input- and program-specific execution trace So then, instead of programming the SM executor ourselves with a specific set of instructions as we did with the mFibonacci SM, the executor of a Generic SM is programmed to read arbitrary instructions encapsulated in some program (depending on the capacity of the SM or the SM's context of application). As mentioned above, each of these programs is initially written, not in a language like Javascript, but in the zkASM language. State machine instructions We continue with the state machine model shown in Figure 1 above; a state machine with two registries $\texttt{A}$ and $\texttt{B}$, and executes computations as per instruction(s) specified in a program. Here is an example of a program containing four instructions, expressed in the zkASM language, $$ \begin{aligned} \begin{array}{|l|c|} \hline \texttt{ } & \bf{Instructions } \text{ }\text{ } \\ \hline \texttt{ } & \mathtt{\$\{getAFreeInput()\} => A} \text{ }\\ \hline \texttt{ } & \mathtt{3 => B} \qquad\qquad\qquad\quad \\ \hline \texttt{ } & \mathtt{:ADD } \qquad\qquad\qquad\quad\quad\text{ }\text{ } \\ \hline \texttt{ } & \mathtt{:END } \qquad\qquad\quad\qquad\text{}\text{ }\text{ } \\ \hline \end{array} \end{aligned} $$ Suppose the state machine starts with the initial state $\big(\texttt{A},\texttt{B}\big) = \big(\texttt{0},\texttt{0} \big)$. The SM executor sequentially executes each instruction as follows, - Firstly, "$\mathtt{\$\{getAFreeInput()\} => A}$" is a request to execute the function $\texttt{getAFreeInput()}$. That is, the executor must get a free input value and move it into register $\texttt{A}$. Note that "free input" simply means the input can be any numeric value. - Secondly, "$\mathtt{3 => B}$" means the executor must move the constant value $\mathtt{3}$ into register $\mathtt{B}$. Since the value $\texttt{3}$ is part of an instruction, it is referred to as the constant of the execution. Also, it is called a constant because, for a given program, it cannot be freely chosen. - Thirdly, "$\mathtt{:ADD }$" instructs the executor to compute the sum of registry value in $\mathtt{A}$ with the one in $\mathtt{B}$, and save the output into register $\mathtt{A}$. - Lastly, "$\mathtt{:END }$" tells the executor to reset the registries $\mathtt{A}$ and $\mathtt{B}$ to their initial values in the next state, and thus achieving the cyclic behaviour. Execution trace In addition to carrying out computations as per instructions in programs, the executor must also generate the trace of all state transitions, called the execution trace. Consider, as an example, the execution trace the executor produces for the above program of four instructions. Suppose the free input value used is 7. The generated execution trace can be depicted in tabular form as shown below. $$ \begin{aligned} \begin{array}{|l|c|c|c|c|c|c|c|}\hline \texttt{ } & \bf{Instructions } \text{ }\text{ }\text{ } & \texttt{FREE} & \texttt{CONST}& \texttt{A}& \mathtt{A'}& \texttt{B}& \mathtt{B'} \\ \hline \texttt{ } & \mathtt{\$\{getAFreeInput()\} => A} \text{ } & \texttt{7} & \texttt{0} & \texttt{0} & \texttt{7} & \texttt{0} & \texttt{0}\\ \hline \texttt{ } & \mathtt{3 => B} \qquad\qquad\qquad\quad & \texttt{0} & \texttt{3} & \texttt{7} & \texttt{7} & \texttt{0} & \texttt{3} \\ \hline \texttt{ } & \mathtt{:ADD } \qquad\qquad\qquad\quad\quad\text{ }\text{ } & \texttt{0} & \texttt{0} & \texttt{7} & \texttt{10} & \texttt{3} & \texttt{3} \\ \hline \texttt{ } & \mathtt{:END } \qquad\qquad\quad\qquad\text{}\text{ }\text{ } & \texttt{0} & \texttt{0} & \texttt{10} & \texttt{0} & \texttt{3} & \texttt{0} \\ \hline \end{array} \end{aligned} $$ This execution trace utilises a total of six columns. Perhaps the use of the columns corresponding to the two registries $\texttt{A}$ and $\texttt{B}$, as well as the columns for the constant $\texttt{CONST}$ and the free input $\texttt{FREE}$, are a bit obvious. But the reason for having the other two columns, $\mathtt{A'}$ and $\mathtt{B'}$, may not be so apparent. The reason there are two extra columns, instead of only four, is the need to capture each state transition in full, and per instruction. The column labelled $\mathtt{A'}$ therefore denotes the next state of the registry $\mathtt{A}$, and similarly, $\mathtt{B'}$ denotes the next state of the registry $\mathtt{B}$. This ensures that each row of the execution trace reflects the entire state transition pertaining to each specific instruction. The execution trace is therefore read row-by-row as follows, - The first row, which is per instruction "$\mathtt{\$\{getAFreeInput()\} => A}$", reads thus: The function $\texttt{getAFreeInput()}$ was executed, by getting the free input value of 7, which is to be moved into the registry $\texttt{A}$ as the next registry value. - The second row, which is per instruction "$\mathtt{3 => B}$", reads as follows: A constant value $\texttt{3}$ is to be moved into the registry $\texttt{B}$ as the next registry value, and the registry $\texttt{A}$ now reflects the value $\texttt{7}$ as expected per previous instruction. - The third row, which is per instruction "$\mathtt{:ADD }$", indicates that: The executor executed the sum of registry values in $\mathtt{A}$ and $\mathtt{B}$ $\big(\text{i.e.},\ \mathtt{7 + 3 = 10}\big)$ and the output $\mathtt{10}$ is to be moved into register $\mathtt{A}$, as the next value of the $\mathtt{A}$ registry. - The last row, which is per instruction "$\mathtt{:END }$", tells us that: The executor has updated the registry value in $\mathtt{A}$ according to the previous instruction, and will reset the registries $\mathtt{A}$ and $\mathtt{B}$ to zeros (their initial values) as their next registry values. Note that, for this specific program, a change in the free input from 7 to another number would obviously yield a different execution trace, yet without violating the first instruction. # plookup.md: The main state machine executor sends various instructions to the secondary state machines within the zkProver. Although secondary state machines specialize in specific types of computations, they frequently use Plookup in their PIL codes to complete tasks mandated by the Main state machine executor. This subsection is part of the generic state machine and its goal is to define Plookup before showing how it is used in PIL verification. What is Plookup? Plookup was described by the original authors in [[GW20]](https://eprint.iacr.org/2020/315.pdf) as a protocol for checking whether values of a committed polynomial, over a multiplicative subgroup H of a finite field $\mathbb{F}$, are contained in a vector $\mathbf{t} \in \mathbb{F}^d$ that represents values of a table $\mathcal{T}$. More precisely, Plookup is used to check if certain evaluations of some committed polynomial are part of some row $\mathbf{t}$ of a lookup table $\mathcal{T}$. One particular use case of this primitive is: checking whether all evaluations of a polynomial $f(x)$, restricted to values of a multiplicative subgroup $\text{H} \subset \mathbb{F}$, fall in a given range $\{ 0 , 1 , \dots , M \}$. i.e., proving that, for every $z \in \text{H}$, we have $\mathbf{f(z) \in \{ 0 , 1 , \dots , M \}}$. Plookup's strategy for soundness depends on a few basic mathematical concepts described below. Notations and preliminaries Sets and multisets Recall that a set is a collection of distinct objects, called elements, where repetitions and order of elements are disregarded. That is, the set $\{ 1 , 3 , 7 \}$ is the same set as $\{ 3 , 7 , 1 , 7 , 1 , 1 \}$. Yet, as multisets, $\{ 1 , 3 , 7 \}$ is not the same as $\{ 3 , 7 , 1 , 7 , 1 , 1 \}.$ It is because multisets take into consideration all instances of an element. Given a multiset $\mathbf{s}$, we define multiplicity to be the number of all instances of an element. So, in the multiset $\{ 3 , 7 , 1 , 7 , 1 , 1 \}$; the element 3 has multiplicity 1, while 7 is of multiplicity 2 , and lastly, 1 has multiplicity 3. A set can therefore be described as a collection of distinct objects where multiplicity and ordering of objects have no significance. Multisets are similar to sets in that they also do not respect the ordering of elements. That is, $\{ 1 , 3 , 1 , 7 \}$ and $\{ 1, 7 , 1 , 3 \}$ represent the same multiset, because they both contain the same elements, each with the same multiplicity. Given multisets $\mathbf{t} = \{ 1 , 3 , 7 \}$ and $\mathbf{s} = \{ 3 , 7 , 1 , 7 , 1 , 1 \}$, $\mathbf{s}$ can be sorted by $\mathbf{t}$ as follows, $\mathbf{s} = \{ 1 , 1 , 1 , 3 , 7 , 7 \}$. That is, "$\mathbf{s}$ sorted by $\mathbf{t}$" means the elements of $\mathbf{s}$ are ordered according to the order they appear in $\mathbf{t}$, without losing their multiplicity. Sorted multisets and set differences Unless otherwise stated, all sorted multisets are henceforth sorted by $\mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers $\{1, 2, 3, \dots\}$. For any given sorted multiset $\mathbf{s} = \{a1 , a2 , \dots , an\}$, define the set of differences of $\mathbf{s}$ as the set of non-zero differences $\{a2 - a1 , a3 - a2 , \dots , an - a{n-1}\}$. That is, zero-differences, $ai - a{i-1} = 0$, are discarded. Take as examples the following sorted multisets: $\mathbf{t} = \{1, 3, 7\}$, $\mathbf{s} = \{1 , 1 , 1 , 3 , 7 , 7\}$ and $\mathbf{r} = \{2, 6, 6, 6, 8\}$. These three multisets have the same set of differences, which is $\{2, 4\}$. Although $\mathbf{r}$ has the same set of differences as $\mathbf{s}$ and $\mathbf{t}$, note that the differences appear in a different order; 4 appeared first then 2. Testing for containment Let $\mathbf{s}$ and $\mathbf{t}$ be ordered multisets, $\mathbf{s} = \{s1 , s2 , \dots , sd\}$ and none of the elements of $\mathbf{t} = \{t1 , t2 , \dots , tn\}$ are repeated. It can be observed that: - If $\mathbf{s} \subset \mathbf{t}$ and $sd = tn$, then $\mathbf{s}$ and $\mathbf{t}$ have the same set of differences and the differences appear in the same order. - If $\mathbf{s}$ and $\mathbf{t}$ have the same set of differences except for the differences' order of appearance, then $\mathbf{s} \not\subset \mathbf{t}$. - If $\mathbf{s}$ and $\mathbf{t}$ have the same set of differences and the differences appear in the same order, then $\mathbf{s} \subset \mathbf{t}$. It follows from these three observations that the criteria for testing containment "$\mathbf{s} \subset \mathbf{t}$" boils down to checking: 1. The equality of the sets of differences, and 2. The order in which the differences ($\mathbf{aj - a{j-1}}$) appear in both multisets are the same. Testing for repeated elements Consider again the ordered multisets: $\mathbf{t} = \{t1 , t2 , \dots , tn\}$ and $\mathbf{s} = \{s1 , s2 , \dots , sd\}$. If $\mathbf{s}$ has some repeated elements, $si = s{i+1}$ for some $1 \leq i \leq n$, then these can be tested by comparing randomized sets of differences. A randomized set of differences related to $\mathbf{s}$ is created by selecting a random field element $\beta \in \mathbb{F}$ and defined as the set $\{si + \beta\cdot s{i+1}\ |\ 1 \leq i < d \}$. So, instead of a pair $(si, s{i+1})$ of repeated elements yielding a difference of zero because $si = s{i+1}$, they yield $$ si + \beta \cdot s{i+1} = (1 + \beta)\cdot si $$ in the setting of randomized set of differences. This property of repeated elements yielding a multiple of $(1 + \beta)$, characterizes repeated elements in ordered multisets. Consequently, when computing randomized set of differences, repeated elements can be identified by these multiples of $(1 + \beta)$. A test can therefore be coined using a grand product argument akin to the one used in PLONKâ€™s permutation argument [[GWC19]](https://eprint.iacr.org/2019/953). Vectors The above concepts defined for multisets apply similarly to vectors, and the Plookup protocol also extends readily to vectors. A vector is a collection of ordered field elements, for some finite field $\mathbb{F}$, and it is denoted by $\mathbf{a} = (a1 , a2 , \dots , an)$. A vector $\mathbf{a} = (a1 , a2 , \dots , an)$ is contained in a vector $\mathbf{b} = (b1 , b2 , \dots , bd)$, denoted by ${\mathbf{a}} \subset {\mathbf{b}}$, if each $ai \in \{ b1 , b2 , \dots , bd \}$ for $i \in \{ 1 , 2 , \dots , n \}$. The vector of differences of a given vector $\mathbf{a} = (a1 , a2 , \dots , an)$ is defined as the vector $\mathbf{a'} = (a2 - a1, a3 - a2, \dots , an - a{n-1})$, which has one less component (or element) compared to $\mathbf{a}$. That is, $|\mathbf{a'}| = n - 1$ because $|\mathbf{a}| = n$. Given a random scalar β in a field $\mathbb{F}$ and a vector $\mathbf{a} = (a1 , a2 , \dots , an)$, a randomized vector of differences of $\mathbf{a}$ is defined as $\mathbf{a'} = (a1 + \beta\cdot a2 , a2 + \beta\cdot a3 , \dots , a{n-1} + \beta\cdot a{n})$. The concatenation of vectors $\mathbf{a} = (a1 , a2 , \dots , an)$ and $\mathbf{b} = (b1 , b2 , \dots , bd)$ is the vector $(\mathbf{a}, \mathbf{b}) = (a1 , a2 , \dots , an , b1 , b2 , \dots , bd)$, and it has $n + d$ components. Plookup protocol The context of Plookup is within a polynomial commitment scheme, where a party $\mathcal{P}$,called the prover, seeks to convince the second party $\mathcal{V}$, called the verifier, that it knows a certain polynomial. The protocol, as described in the Plookup paper, mentions a third party called the Ideal party, denoted by $\mathcal{I}$, which is also called the trusted party. It is this trusted party who is responsible for generating all the proof-verification system parameters. Among these parameters are the so-called preprocessed polynomials. In the case of a non-interactive proof-verification system, the trusted party is equipped so as to act as an intermediary between the prover and the verifier. 1. The protocol starts with the generation of preprocessed polynomials $\{t\}$ and describing them as a Lookup Table. 2. Prover submits messages in the form of polynomials $\{f\}$ expressed either as multisets or vectors. 3. The verifier also sends randomly selected field elements $\{\alphai\} \subset \mathbb{F}$, called challenges. 4. The prover then evaluates a special polynomial $Z[X]$ on the $\alphai$'s and sends them to $\mathcal{I}$. 5. The verifier subsequently requests $\mathcal{I}$ to check if certain polynomial identities $\{F \equiv G\}$ hold true. 6. The verifier accepts the prover's submissions as true if all identities hold true, otherwise it rejects. The polynomials F and G in the polynomial identities $\{F \equiv G\}$ are bi-variate polynomials in β and γ, related to randomized sets of differences associated with $\{f\}$ and $\{t\}$. They are defined in terms of grand product expressions seen below, $$ \begin{aligned} F(\beta, \gamma) := (1+\beta)^n \cdot \prod{i \in [n]} (\gamma + fi) \prod{i \in [d-1]} (\gamma (1+\beta) + ti + \beta t{i+1}) \\ G(\beta, \gamma) := \prod{i\in[n+d-1]}(\gamma(1+\beta) + si + \beta s{i+1}) \end{aligned} $$ where β and γ are the randomly selected field elements. The Plookup protocol boils down to proving that the two polynomials F and G are the same by comparing vectors of their evaluations and multiplicities of elements in those vectors. Simply put, it checks if two polynomials are the same up to multiplicities of elements in their witness vectors. !!!info Plookup optimization strategies - Using randomized sets of differences. - Soundness is backed by the Schwarz-Zippel lemma. - Deploying the grand product argument which describes the polynomials F and G appearing in the polynomial identities. - Concatenation of witness vectors with the LUT vectors. Plookup in PIL The test for whether the polynomial identities $\{F \equiv G\}$ hold true, is based on the following lemma (viz. Claim 3.1 as stated and proved in pages 4 and 5 of the Plookup paper). $\mathbf{Lemma\ 1}.$ The polynomial identities $\{F \equiv G\}$ hold true if, and only if, $f \subset t$ and $s = (f,t)$, which is the concatenation of f and t, is sorted by t. A Plookup applies to Polynomial Commitment Scheme settings, where the polynomials are expressed as vectors; t, f and $s = (f,t)$. It is therefore a test of inclusion, denoted by $f \subset t$. In the zkEVM setting, Plookup typically appears in PIL code as: $$ \{f1, \dots , fm \} \texttt{ in } \{t1, \dots , tm \} $$ where "$\texttt{in}$" is a PIL keyword for set inclusion, $f \subset t$. Application of Plookup in PIL In the generic state machine, recall polynomials that define instructions; $\texttt{CONST}$, $\texttt{inA}$, $\texttt{inB}$, $\texttt{inFREE}$, $\texttt{setA}$, $\texttt{setB}$, $\texttt{JMP}$, $\texttt{JMPZ}$, $\texttt{offset}$ and $\texttt{line}$. The polynomials; $\texttt{A}$, $\texttt{B}$, $\texttt{FREE}$ and $\texttt{invOp}$; are only used during execution, and therefore do not form part of the instructions. When verifying correct execution, there is a need to check if the combination of committed polynomials, $$ \{\texttt{CONST}, \texttt{inA}, \texttt{inB}, \texttt{inFREE}, \texttt{setA}, \texttt{setB}, \texttt{JMP}, \texttt{JMPZ}, \texttt{offset}, \texttt{zkPC} \} $$ is one of the valid combinations of the ROM instructions, $$ \{ \texttt{Rom.CONST}, \texttt{Rom.inA},\ \texttt{Rom.inB},\ \texttt{Rom.inFREE},\ \texttt{Rom.setA},\ \texttt{Rom.setB},\ \texttt{Rom.JMP},\\ \texttt{Rom.JMPZ},\ \texttt{Rom.offset},\ \texttt{Rom.line} \}. \\ $$ PIL uses Plookup to carry out this check. This appears at the last three lines of the PIL code for the generic state machine, shown in the figure below. Recall that the ROM's polynomial counterpart to the $\texttt{zkPC}$ is the $\texttt{line}$. !Plookup in the PIL Code for the generic state machine Role in zkEVM Plookup instructions appear in the PIL codes of state machines in the zkEVM. Plookup is used by some state machines to ensure that certain output values are included in a given lookup table. Plookup is useful when set inclusion must be verified. The diagram below depicts the extensive role that Plookup plays in the zkProver's secondary state machines. !Plookup and the zkProver state machines # program-counter.md: Checking sequence of instructions In order to keep track of which line of the program is currently being executed, a new registry called "program counter" is added to the state machine. We denote it by $\texttt{zkPC}$ because it is verified in a zero-knowledge manner. The $\texttt{zkPC}$ is therefore a new column of the execution trace and it contains, at each clock, the line in the zkASM program of the instruction being executed. !Figure 7: A state machine with a program counter In addition to the $\texttt{JMPZ(addr)}$, an unconditional jump instruction $\texttt{JMP(addr)}$ is allowed in the zkASM program. Unlike the $\texttt{JMPZ(addr)}$ instruction, when the state machine executes $\texttt{JMP(addr)}$, it must jump to the line $\texttt{addr}$, irrespective of what the value of $\texttt{op}$ is. Program counter constraint related to JMP This is how the $\texttt{JMP(addr)}$ instruction is implemented: A new selector called $\texttt{JMP}$ is added, as a column to the execution trace. And, the program counter $\texttt{zkPC}$ now uses the following identity to keep track of the correct line of the assembly program to be executed next; $$ \mathtt{zkPC' = (zkPC+1)+JMP \cdot \big(addrâˆ’(zkPC+1)\big)} \tag{Eqn 0} $$ $\texttt{JMP}$ therefore acts as a 'flag' where; - $\mathtt{zkPC' = (zkPC+1)+ 0 \cdot \big(addrâˆ’(zkPC+1)\big) = zkPC+1}$, if $\texttt{JMP}$ is not activated (i.e., if $\texttt{JMP}$ is $\mathtt{0}$), or - $\mathtt{zkPC' = (zkPC+1)+ 1 \cdot \big(addrâˆ’(zkPC+1)\big) = addr}$, if $\texttt{JMP}$ is activated (i.e., if $\texttt{JMP}$ is $\mathtt{1}$). Note that execution continues with the next chronological line of instruction $\texttt{zkPC+1}$ when $\texttt{JMP}$ is $\mathtt{0}$, but otherwise proceeds to execute the instruction in line number $\texttt{addr}$. Program counter constraint related to JMPZ The $\texttt{JMPZ(addr)}$ similarly needs a selector to be added as an extra column to the execution trace, call it $\texttt{JMPZ}$. Since the $\texttt{JMPZ(addr)}$ instruction is executed only on condition that $\texttt{op}$ is zero, we introduce a flag called $\mathtt{isZero}$, such that; $\mathtt{isZero = 1}$ if $\texttt{op = 0}$, and $\mathtt{isZero = 0}$ if $\mathtt{op \not= 0}$. This means $\texttt{op}$ and $\mathtt{isZero}$ satisfy the following constraint, $$ \mathtt{isZero \cdot op = 0}. \qquad\qquad\quad \tag{Eqn 1} $$ Note that $\texttt{op}$ is a field element, and every non-zero element of the prime field $\mathbb{F}p$ has an inverse. That is, $\texttt{op}^{-1}$ exists if and only if $\mathtt{op \not= 0}$. The following constraint together with Eqn 1 can be tested to check whether $\texttt{op}$ and $\mathtt{isZero}$ are as required, $$ \mathtt{isZero\ := (1 âˆ' op \cdot op^{âˆ'1})} \qquad \tag{Eqn 2} $$ Since $\mathtt{JMPZ = 1}$ only if $\texttt{JMPZ(addr)}$ is the very instruction in the line currently being executed, and $\mathtt{zkPC' = addr}$ only if $\mathtt{isZero = 1}$. It follows that the factor $(\mathtt{JMPZ \cdot isZero})$ $\mathtt{= 1}$ only if both $\texttt{JMPZ}$ $\mathtt{= 1}$ and $\mathtt{isZero}$ $\mathtt{= 1}$ hold true. Hence, the following constraint enforces the correct sequence of instructions when $\texttt{JMPZ}$ is executed; $$ \mathtt{zkPC' = (zkPC+1)+(JMPZ \cdot isZero) \cdot \big(addrâˆ'(zkPC+1)\big)}\mathtt{\ } \tag{Eqn 3} $$ In order to ascertain correct execution of the $\texttt{JMPZ(addr)}$ instruction, it suffices to check the above three constraints; Eqn 1, Eqn 2 and Eqn 3. Note that $\mathtt{op^{âˆ'1}}$ is not part of any instruction. The evaluations of $\mathtt{op^{âˆ'1}}$ therefore have to be included in the execution trace as a committed polynomial, and thus enables checking the first identity, Eqn 1. Most importantly, we rename the polynomial $\mathtt{op^{âˆ'1}}$ as $\texttt{invOp}$. Program counter constraint related to JMP and JMPZ All four constraints, Eqn 0, Eqn 1, Eqn 2 and Eqn 3 can be merged as follows; $$ \begin{aligned} &\mathtt{isZero\ := (1 âˆ' op \cdot op^{âˆ'1})},\qquad\text{ }\qquad \\ &\mathtt{isZero \cdot op = 0},\qquad\qquad\qquad\qquad\qquad\qquad\quad \\ &\mathtt{doJMP := JPMZ \cdot isZero + JMP}, \qquad\qquad\qquad\qquad\quad \\ &\mathtt{zkPCâ€² = (zkPC+1)+doJMP \cdot \big(addrâˆ'(zkPC+1)\big)} &\end{aligned} $$ Define "$\texttt{addr}$" as an intermediate polynomial, $$ \mathtt{ addr := offset}, \tag{Eqn } $$ where $\texttt{offset}$ is provided as part of the jump instructions, $\texttt{JMPZ(addr)}$ and $\texttt{JMP(addr)}$. We now have the following new columns in the execution trace: $$ \begin{array}{|c|c|c|c|c|c|}\hline \mathtt{zkPC} & \mathtt{JMP} & \mathtt{JMPZ} & \mathtt{invOp} & \mathtt{offset}\\\hline \quad\mathtt{\dots\ } & \mathtt{\dots } & \mathtt{\dots} & \mathtt{\dots} & \mathtt{\dots}\\\hline \quad\mathtt{\dots} & \mathtt{\dots} & \mathtt{\dots} &\mathtt{\dots} & \mathtt{\dots}\\\hline \quad\mathtt{\dots} & \mathtt{\dots} & \mathtt{\dots} & \mathtt{\dots} & \mathtt{\dots}\\\hline \quad\mathtt{\dots} & \mathtt{\dots} & \mathtt{\dots} & \mathtt{\dots} & \mathtt{\dots}\\ \end{array} $$ With the intermediate definition in Eqn , we are preparing the path to expanding the $\texttt{addr}$ definition, for example, to include other sources (apart from instructions) from which to get the destination address in jumps. Correct program ending with a loop We have previously resorted to ending our program by repeating the "$\texttt{:END}$" instruction. This was not the best solution. To properly finish our programs, we need to ensure that the execution trace generated from the assembly is cyclic. We therefore utilise a loop while the current $\texttt{line}$ is less than $\mathtt{N-1}$, a step before the last step of the execution trace. That is, before jumping back to the start, with the $\texttt{JMP(start)}$ instruction, the current $\texttt{line}$ is checked with the function "$\texttt{beforeLast()}$". So, we query the executor for when the execution trace is at its last but one row, with the following line of code, $beforeLast() :JMPZ(finalWait); The assembly program uses labels instead of explicit line numbers, hence conforming to how assembly programs are typically written. Labels are convenient because with them, the assembly compiler can take care of computing actual line numbers. For instance, as in Figure 8 below; $\mathtt{start}$ is line $\mathtt{0}$ and $\mathtt{finalWait}$ is line $\mathtt{5}$. !Figure 8: Ending a program with a loop Commentary on the execution trace The intention here to give a step-by-step commentary on the execution trace for the Assembly code in Figure 8 above, for a free input equal to $\mathtt{3}$ and a trace size (interpolation size) equal to $\texttt{N = 8}$. We show the values corresponding to each $\texttt{step}$ of the execution trace, particularly for, - The committed polynomials that form part of the instruction; $\texttt{CONST}$, $\texttt{offset}$, $\texttt{JMP}$, $\texttt{JMPZ}$, $\texttt{setB}$, $\texttt{setA}$, $\texttt{inFREE}$, $\texttt{inB}$, $\texttt{inA}$, $\texttt{zkPC}$ and $\mathtt{zkPC'}$. - The committed polynomials that are only part of the trace; $\texttt{FREE}$, $\texttt{A}$, $\mathtt{A'}$, $\texttt{B}$, $\mathtt{B'}$ and $\texttt{invOp}$. - The intermediate polynomials $\texttt{Im}$, which are just intermediate definitions; $\texttt{op}$, $\texttt{isZero}$ and $\texttt{doJMP}$. Step 0: "getAFreeInput()=>A" Since the instruction here at $\texttt{Step}$ $\mathtt{0}$ (corresponding to $\texttt{line}$ $\texttt{0}$ of the Assembly code) requires a free input $\mathtt{3}$ to be taken and moved into the registry $\mathtt{A}$. The values of the involved polynomials are therefore; $$ \mathtt{inFREE = 1},\ \mathtt{FREE = 3},\ \mathtt{setA = 1},\ \mathtt{A' = 3}\ \texttt{and}\ \mathtt{CONST = 0}. $$ As a result, $$ \mathtt{op\ =\ inA \cdot A\ +\ inB \cdot B\ +\ inFREE \cdot FREE\ +\ CONST = 0 \cdot A\ +\ 0 \cdot B\ +\ 1 \cdot 3\ +\ 0\ = 3} $$ and $\mathtt{invOp = 3^{-1}}$. Regarding the program counter, first note that initially $\mathtt{zkPC = 0}$. And, since $\mathtt{op = 3}$ and $\mathtt{invOp = 3^{-1}}$, then $$ \mathtt{isZero\ := (1 âˆ' op \cdot op^{âˆ'1}) = (1 âˆ' 3 \cdot 3^{âˆ'1}) = 0}.\qquad\qquad\qquad\qquad\text{ }\qquad \\ $$ Also, since there are no jumps in the instruction, $\mathtt{JMP = 0}$ and $\mathtt{JMPZ = 0}$, and therefore $\mathtt{zkPCâ€² = zkPC + 1}$. We use Eqn 4 to check whether $\mathtt{zkPCâ€² = zkPC + 1 = 0+1 = 1}$. First, note that, $$ \mathtt{doJMP := JPMZ \cdot isZero + JMP = 0 \cdot 0 + 0 = 0}. \qquad\qquad\qquad\qquad\qquad \\ $$ This means, according to Eqn 4, $$ \mathtt{zkPCâ€² = (zkPC+1)+doJMP \cdot \big(addrâˆ'(zkPC+1)\big) = (0+1) + 0 \cdot \big(addrâˆ'(0+1) \big) = 0+1 = 1}. $$ Step 1: "âˆ'3=>B, âˆ'3=>B" In this step, a constant $\mathtt{CONST = -3}$ is moved into the $\texttt{B}$ registry. Hence $\mathtt{setB = 1}$, $\mathtt{B' = -3}$, but $\mathtt{inB = 0}$, $\mathtt{inA = 0}$ and $\mathtt{inFREE = 0}$. This yields, $$ \mathtt{op =\ inA \cdot A\ +\ inB \cdot B\ +\ inFREE \cdot FREE\ +\ CONST\ =\ 0 \cdot A\ +\ 0 \cdot B\ +\ 0 \cdot FREE\ + (-3) = -3} $$ and $\mathtt{invOp = (-3)^{-1}}$. So, $\mathtt{\ isZero\ := (1 âˆ' op \cdot op^{âˆ'1}) = \big(1 âˆ' (-3) \cdot (-3)^{âˆ'1}\big) = 0}$. Again, the absence of jumps in the instruction means, $\mathtt{JMP = 0}$ and $\mathtt{JMPZ = 0}$. And therefore $\mathtt{zkPCâ€² = zkPC + 1}$. Again, we use Eqn 4 to check whether $\mathtt{zkPCâ€² = zkPC + 1 = 1+1 = 2}$. Note that, $$ \mathtt{doJMP := JPMZ \cdot isZero + JMP = 0 \cdot 0 + 0 = 0}. \qquad\qquad\qquad\qquad\qquad \\ $$ Since $\mathtt{zkPC = 1}$, the next value of the program counter (according to Eqn 4) must therefore be, $$ \mathtt{zkPCâ€² = (zkPC+1)+doJMP \cdot \big(addrâˆ'(zkPC+1)\big) = (1+1) + 0 \cdot \big(addrâˆ'(1+1) \big) = 2}. $$ Here is the execution trace thus far; $$ \small \begin{aligned} \begin{array}{|l|c|c|c|c|c|c|c|c|c|} \hline \texttt{step} & \bf{instructions} & \mathtt{CONST} & \mathtt{offset} & \mathtt{JMP} & \mathtt{JMPZ} & \mathtt{setB} & \mathtt{setA} & \mathtt{inFREE} & \mathtt{inB} & \mathtt{inA} & \mathtt{zkPC} & \mathtt{zkPC'} \\ \hline \quad\texttt{0} & \mathtt{{getAFreeInput()} => A}\text{ } & \quad\mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{1} \\ \hline \quad\texttt{1} & \mathtt{-3 => B}\text{ }\qquad\qquad\qquad\text{ }\text{ } & \ \mathtt{-3} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{2} \\ \hline \end{array} \end{aligned} $$ $$ \small \begin{aligned} \begin{array} {|l|c|c|c|c|c|c|c|c|}\hline \mathtt{op} & \mathtt{isZero} & \mathtt{doJMP} & \mathtt{FREE} & \mathtt{A} & \mathtt{A'} & \mathtt{B} & \mathtt{B'} & \mathtt{invOp} \\\hline \text{ }\mathtt{3} & \mathtt{0} & \mathtt{0} & \quad\mathtt{3} & \mathtt{0} & \mathtt{3} & \mathtt{0} & \mathtt{0\ } & \mathtt{3^{-1}} \\\hline \mathtt{-3} &\mathtt{0} & \mathtt{0} & \quad\mathtt{0} &\mathtt{3} & \mathtt{3} & \mathtt{0} & \mathtt{-3\ } & \mathtt{(-3)^{-1}}\\\hline \end{array} \end{aligned} $$ Step 2: ": ADD" Here the sum of the registry values $\mathtt{A = 3}$ and $\mathtt{B = -3}$ is computed, and the result is moved into the registry $\mathtt{A}$. That is, $\mathtt{A' = 3 + (-3) = 0}$ and $\mathtt{setA = 1}$. Also, $\mathtt{inA = 1}$, $\mathtt{inB = 1}$ and $\mathtt{inFREE = 0}$. These values yield the following value of $\mathtt{op}$, $$ \mathtt{op\ = inA \cdot A + inB \cdot B + inFREE \cdot FREE + CONST = 1 \cdot 3 + 1 \cdot (-3) + 0 \cdot FREE + 0 = 0}. $$ So, $\mathtt{invOp}$ is set to a randomly chosen non-zero $\mathtt{\alpha}$ in $\mathbb{F}p$, used to pass the identities related to $\texttt{isZero}$. And, $\mathtt{isZero := (1 âˆ' op \cdot invOp) = \big(1 âˆ' 0\cdot \alpha \big) = 1}$. Note that there are no jumps in the instruction, so $\mathtt{JMP = 0}$ and $\mathtt{JMPZ = 0}$. And therefore $\mathtt{zkPCâ€² = zkPC + 1}$. In order to verify that $\mathtt{zkPCâ€² = zkPC + 1 = 2 + 1 = 3}$, we use the constraints given as Eqn 4 above. Firstly, check $\mathtt{doJMP}$ as follows, $$ \mathtt{doJMP\ :=\ JPMZ \cdot isZero + JMP\ = 0 \cdot 1\ +\ 0\ =\ 0}. \qquad\qquad\qquad\qquad\qquad \\ $$ Then, $$ \mathtt{zkPCâ€² = (zkPC+1)+doJMP \cdot \big(addrâˆ'(zkPC+1)\big) = (2+1) + 0 \cdot \big(addr - (2+1)\big) = 3}. $$ The program counter therefore moves to the subsequent line of instruction. That is, the next instruction to be executed must the one in $\texttt{line}$ $\texttt{3}$ of the Assembly code. Step 3: "A :JMPZ(finalWait)" According this instruction, the executor has to jump on condition that $\texttt{A}$ is $\mathtt{0}$, otherwise there is no jump. The polynomials immediately involved in this instruction are $\mathtt{inA}$ and $\mathtt{JMPZ}$. Therefore, $\mathtt{inA = 1}$ and $\mathtt{JMPZ = 1}$. As mentioned above, the implicit address label "$\mathtt{finalWait}$" is computed by the Assembly compiler, and in this example, that address is $\mathtt{line}$ $\mathtt{5}$. That is, $\mathtt{offset = 5}$. It follows that $\mathtt{zkPCâ€² = 5}$. Note that, $\mathtt{inB = 0}$, $\mathtt{inFREE = 0}$ and $\mathtt{CONST = 0}$. Therefore, $$ \mathtt{op = inA \cdot A + inB \cdot B + inFREE \cdot FREE + CONST = 1 \cdot 0 + 0 \cdot (-3) + 0 \cdot FREE + 0 = 0}. $$ Consequently, $\mathtt{isZero := (1 âˆ' op \cdot invOp) = \big(1 âˆ' 0 \big) = 1}$. And since there are no unconditional jumps, $\mathtt{JMP = 0}$. We use Eqn 4 to check whether $\mathtt{zkPCâ€² = 5}$. Note that, $$ \mathtt{doJMP := JPMZ \cdot isZero + JMP = 1 \cdot 1 + 0 = 1}. \qquad\qquad\qquad\qquad\qquad \\ $$ The next value of the program counter, according to Eqn 4, is $$ \mathtt{zkPCâ€² = (zkPC+1)+doJMP \cdot \big(addrâˆ'(zkPC+1)\big) = (3+1) + 1 \cdot \big(5âˆ'(3+1)\big) = 4 + (5-4) = 5.} $$ Step 4: "{beforeLast()} :JMPZ(finalWait)" The $\texttt{beforeLast()}$ function, which keeps track of the number of steps being executed, reads the current step-number as a free input. Since the execution trace is currently at step $\mathtt{4}$ and not $\mathtt{6}$, then the executor returns a zero. And thus, $\mathtt{inFREE = 1}$ and $\mathtt{JMPZ = 1}$ but $\mathtt{inA = 0}$, $\mathtt{inB =0}$, $\mathtt{FREE = 0}$ and $\mathtt{CONST = 0}$. Consequently, $$ \mathtt{op = inA \cdot A + inB \cdot B + inFREE \cdot FREE + CONST = 0 \cdot A + 0 \cdot B + 1 \cdot 0 + 0 = 0}. $$ Therefore $\mathtt{isZero := (1 âˆ' op \cdot invOp) = (1 âˆ' 0 \cdot \alpha) = 1}$. Hence according to $\texttt{JMPZ(finalWait)}$, a jump is executed. This means the executor must jump to the $\mathtt{offset = 5}$ address, as computed by the Assembly compiler. It follows that $\mathtt{zkPCâ€²}$ must be $\mathtt{5}$. Let us use Eqn 4 to check if indeed $\mathtt{zkPCâ€² = 5}$. We first note that, there are no unconditional jumps, so $\mathtt{JMP = 0}$. And, $$ \mathtt{doJMP := JPMZ \cdot isZero + JMP = 1 \cdot 1 + 0 = 1}. \qquad\qquad\qquad\qquad\qquad \\ $$ The next value of the program counter is given by, $$ \mathtt{zkPCâ€² = (zkPC+1)+doJMP \cdot \big(addrâˆ'(zkPC+1)\big) = (5+1) + 1 \cdot \big(5âˆ'(5+1)\big) = 6 + (5-6) = 5.} $$ The execution trace is currently as follows, $$ \small \begin{aligned} \begin{array} {|l|c|c|c|c|c|c|c|c|}\hline \texttt{step} & \bf{instructions} & \mathtt{CONST} & \mathtt{offset} & \mathtt{JMP} & \mathtt{JMPZ} & \mathtt{setB} & \mathtt{setA} & \mathtt{inFREE} & \mathtt{inB} & \mathtt{inA} & \mathtt{zkPC} & \mathtt{zkPC'} \\\hline \quad\texttt{0} & \mathtt{{getAFreeInput()} => A}\quad\qquad\qquad & \quad\mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{1}\\\hline \quad\texttt{1} & \mathtt{-3 => B}\qquad\qquad\qquad\qquad\qquad\qquad\text{ }\text{ } & \ \mathtt{-3} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{2}\\\hline \quad\texttt{2} & \mathtt{:ADD}\qquad\quad\qquad\qquad\qquad\qquad\qquad\ & \quad\mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{1} & \mathtt{1} & \mathtt{2} & \mathtt{3}\\\hline \quad\texttt{3} & \mathtt{A \qquad :JMPZ(finalWait)}\qquad\qquad & \quad\mathtt{0} & \mathtt{5} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{3} & \mathtt{5}\\\hline \quad\texttt{4} & \mathtt{\{beforeLast()\}\quad:JMPZ(finalWait)} & \quad\mathtt{0} & \mathtt{5} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{5} & \mathtt{5}\\\hline \end{array} \end{aligned} $$ $$ \small \begin{aligned} \begin{array}{|l|c|c|c|c|c|c|c|c|}\hline \mathtt{op} & \mathtt{isZero} & \mathtt{doJMP} & \mathtt{FREE} & \mathtt{A} & \mathtt{A'} & \mathtt{B} & \mathtt{B'} & \mathtt{invOp}\\\hline \text{ }\mathtt{3} & \mathtt{0} & \mathtt{0} & \quad\mathtt{3} & \mathtt{0} & \mathtt{3} & \mathtt{0} & \mathtt{0\ } & \mathtt{3^{-1}}\\\hline \mathtt{-3} &\mathtt{0} & \mathtt{0} & \quad\mathtt{0} &\mathtt{3} & \mathtt{3} & \mathtt{0} & \mathtt{-3\ } & \mathtt{(-3)^{-1}}\\\hline \text{ }\mathtt{0} & \mathtt{1} & \mathtt{0} & \quad\mathtt{0} & \mathtt{3} & \mathtt{0} & \mathtt{-3\ } & \mathtt{-3\ } & \mathtt{\alpha}\\\hline \text{ }\mathtt{0} & \mathtt{1} & \mathtt{1} & \quad\mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{-3\ } & \mathtt{-3\ } & \mathtt{\alpha}\\\hline \text{ }\mathtt{0} & \mathtt{1} & \mathtt{1} & \quad\mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{-3\ } & \mathtt{-3} & \mathtt{\alpha}\\\hline \end{array} \end{aligned} $$ Step 5: "{beforeLast()} :JMPZ(finalWait)" As seen in Step 4, the $\texttt{beforeLast()}$ function checks if the execution trace is currently at Step 6. Since the current step is $\mathtt{5}$ and not $\mathtt{6}$, the executor returns a zero. Similarly, $\mathtt{inFREE = 1}$ and $\mathtt{JMPZ = 1}$ but $\mathtt{inA = 0}$, $\mathtt{inB =0}$, $\mathtt{FREE = 0}$ and $\mathtt{CONST = 0}$. As a result, $$ \mathtt{op = inA \cdot A + inB \cdot B + inFREE \cdot FREE + CONST = 0 \cdot A + 0 \cdot B + 1 \cdot 0 + 0 = 0}, $$ which means $\mathtt{FREE = 0}$ and $\mathtt{isZero \ := (1 âˆ' op \cdot invOp)\ = (1 âˆ' 0 \cdot \alpha) = 1}$. So, again $\texttt{JMPZ(finalWait)}$ gets executed. The absence of unconditional jumps means $\mathtt{JMP = 0}$, while $\mathtt{JMPZ = 1}$. Since $\mathtt{offset = 5}$, the program counter, $\mathtt{zkPCâ€²}$, must in the next step be $\mathtt{5}$. We verify this by first checking: $$ \mathtt{doJMP := JPMZ \cdot isZero + JMP = 1 \cdot 1 + 0 = 1}, \qquad\qquad\qquad\qquad\qquad \\ $$ and use Eqn 4, $$ \mathtt{zkPCâ€² = (zkPC+1)+doJMP \cdot \big(addrâˆ'(zkPC+1)\big) = (5+1) + 1 \cdot \big(5âˆ'(5+1)\big) = 6 + (5-6) = 5.} $$ Step 6: "{beforeLast()} :JMPZ(finalWait)" In this case, the current step is the last but one step. That is, the $\texttt{beforeLast()}$ function holds true, and hence the executor must return a $\mathtt{1}$. So, $\mathtt{inFREE = 1}$ and $\mathtt{JMPZ = 1}$ while $\mathtt{inA = 0}$, $\mathtt{inB =0}$ and $\mathtt{CONST = 0}$. Then, $$ \mathtt{op = inA \cdot A + inB \cdot B + inFREE \cdot FREE + CONST = 0 \cdot A + 0 \cdot B + 1 \cdot 1 + 0 = 1}. $$ This means $\mathtt{FREE = 1}$ and $\mathtt{isZero \ := (1 âˆ' op \cdot invOp)\ = (1 âˆ' 1 \cdot 1) = 0}$. And, this time $\texttt{JMPZ(finalWait)}$ is not executed, implying the next program counter, $\mathtt{zkPCâ€² = zkPC + 1}$. Since there are no jumps in this step, $\mathtt{JMP = 0}$ and $\mathtt{JMPZ = 0}$, yielding $$ \mathtt{doJMP := JPMZ \cdot isZero + JMP = 0 \cdot 1 + 0 = 0}, \\ $$ and with a quick verification using Eqn 4, we obtain $$ \mathtt{zkPCâ€² = (zkPC+1)+doJMP \cdot \big(addrâˆ'(zkPC+1)\big) = (5+1) + 0 \cdot \big(addr âˆ'(5+1)\big) = 6.} $$ Step 7: "0=>A,BÂ :JMP(start)" The aim with this instruction is to complete the execution trace such that it is of the correct size (which is $\mathtt{8}$ in this example). It ends the execution by setting $\texttt{A}$ and $\texttt{B}$ to zero, and jumps to $\mathtt{start}$, which is line $\mathtt{0}$. And thus, $\mathtt{zkPC'}$ must be $\mathtt{0}$. Hence, $\mathtt{setA = 1}$, $\mathtt{setB = 1}$ and $\mathtt{JMP = 1}$ but $\mathtt{inFREE = 0}$, $\mathtt{inA =0}$, $\mathtt{inB =0}$ and $\mathtt{CONST = 0}$. Consequently, $$ \mathtt{op = inA \cdot A + inB \cdot B + inFREE \cdot FREE + CONST = 0 \cdot 0 + 0 \cdot (-3) + 0 \cdot 1 + 0 = 0}. $$ Therefore $\mathtt{isZero := (1 âˆ' op \cdot invOp) = (1 âˆ' 0 \cdot \alpha) = 1}$. There are no conditional jumps, so $\mathtt{JMPZ = 0}$. Then, as a consequence of this, $$ \mathtt{doJMP := JPMZ \cdot isZero + JMP = 0 \cdot 1 + 1 = 1}. \\ $$ The constraint in Eqn 4 verifies the next value of the program counter as follows, $$ \mathtt{zkPCâ€² = (zkPC+1)+doJMP \cdot \big(addrâˆ'(zkPC+1)\big)\ =\ (6+1)\ +\ 1 \cdot \big(0 âˆ'(6+1)\big)\ =\ 0.} $$ This instruction, as the last step the Assembly program, achieves two things; Firstly, the program ends correctly with the specified size of the execution trace. Secondly, resetting $\texttt{A}$, $\texttt{B}$ and $\texttt{zkPC}$ to zero causes the execution trace to attain cyclicity. See the complete execution trace below, $$ \small \begin{aligned} \begin{array}{|l|c|c|c|c|c|c|c|c|}\hline \texttt{step} & \bf{instructions} & \mathtt{CONST} & \mathtt{offset} & \mathtt{JMP} & \mathtt{JMPZ} & \mathtt{setB} & \mathtt{setA} & \mathtt{inFREE} & \mathtt{inB} & \mathtt{inA} & \mathtt{zkPC} & \mathtt{zkPC'} \\\hline \quad\texttt{0} & \mathtt{{getAFreeInput()} => A}\quad\qquad\qquad & \quad\mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{1}\\\hline \quad\texttt{1} & \mathtt{-3 => B}\qquad\qquad\qquad\qquad\qquad\qquad\text{ }\text{ } & \ \mathtt{-3} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{2}\\\hline \quad\texttt{2} & \mathtt{:ADD}\qquad\quad\qquad\qquad\qquad\qquad\qquad\ & \quad\mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{1} & \mathtt{1} & \mathtt{2} & \mathtt{3}\\\hline \quad\texttt{3} & \mathtt{A \qquad :JMPZ(finalWait)}\qquad\qquad & \quad\mathtt{0} & \mathtt{5} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{3} & \mathtt{5}\\\hline \quad\texttt{4} & \mathtt{\{beforeLast()\}\quad:JMPZ(finalWait)} & \quad\mathtt{0} & \mathtt{5} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{5} & \mathtt{5}\\\hline \quad\texttt{5} & \mathtt{\{beforeLast()\}\quad:JMPZ(finalWait)} & \quad\mathtt{0} & \mathtt{5} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{5} & \mathtt{5}\\\hline \quad\texttt{6} & \mathtt{\{beforeLast()\}\quad:JMPZ(finalWait)} & \quad\mathtt{0} & \mathtt{5} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{5} & \mathtt{6}\\\hline \quad\texttt{7} & \mathtt{0 => A,B \quad\qquad\ \ :JMP(start)}\qquad & \quad\mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{1} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{6} & \mathtt{0}\\\hline \end{array} \end{aligned} $$ $$ \small \begin{aligned} \begin{array}{|l|c|c|c|c|c|c|c|c|}\hline \mathtt{op} & \mathtt{isZero} & \mathtt{doJMP} & \mathtt{FREE} & \mathtt{A} & \mathtt{A'} & \mathtt{B} & \mathtt{B'} & \mathtt{invOp}\\\hline \text{ }\mathtt{3} & \mathtt{0} & \mathtt{0} & \quad\mathtt{3} & \mathtt{0} & \mathtt{3} & \mathtt{0} & \mathtt{0\ } & \mathtt{3^{-1}}\\\hline \mathtt{-3} &\mathtt{0} & \mathtt{0} & \quad\mathtt{0} &\mathtt{3} & \mathtt{3} & \mathtt{0} & \mathtt{-3\ } & \mathtt{(-3)^{-1}}\\\hline \text{ }\mathtt{0} & \mathtt{1} & \mathtt{0} & \quad\mathtt{0} & \mathtt{3} & \mathtt{0} & \mathtt{-3\ } & \mathtt{-3\ } & \mathtt{\alpha}\\\hline \text{ }\mathtt{0} & \mathtt{1} & \mathtt{1} & \quad\mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{-3\ } & \mathtt{-3\ } & \mathtt{\alpha}\\\hline \text{ }\mathtt{0} & \mathtt{1} & \mathtt{1} & \quad\mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{-3\ } & \mathtt{-3} & \mathtt{\alpha}\\\hline \text{ }\mathtt{0} &\mathtt{1} & \mathtt{1} & \quad\mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{-3\ } & \mathtt{-3\ } & \mathtt{\alpha}\\\hline \text{ }\mathtt{1} & \mathtt{0} & \mathtt{0} & \quad\mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{-3\ } & \mathtt{-3\ } & \mathtt{1}\\\hline \text{ }\mathtt{0} & \mathtt{1} & \mathtt{1} & \quad\mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{-3\ } & \mathtt{0} & \mathtt{\alpha}\\\hline \end{array} \end{aligned} $$ In conclusion, our Assembly program uses jumps to create a loop in its execution. That is, at the right point of the execution, when the registry values of interest have been attained, the executor retains these values until the execution trace is in the last but one row. This way, the result of the computations can be reflected in the last position of the registry $\mathtt{A}$, in the last row of the execution trace, which is the output of the mFibonacci state machine. Note that in cases where the jumps are such that $\mathtt{offset}$ is one of the previous addresses, the length of the execution trace can turn out to be much bigger than the final value of the program counter. i.e., $\mathtt{zkPC \leq 2^N - 1}$ where $\mathtt{2^N -1}$ is the degree of the state machine polynomials. Publics placed at known steps Regarding the $\texttt{publics}$, it is best to place these at specific and known steps. Notice that, in the above example, the final loop added to Assembly program repeats the values of the registries $\mathtt{A}$ and $\mathtt{B}$ until they are reset to Step $\mathtt{0}$. So if, for example, the execution result or state machine output is the state of registry $\mathtt{A}$ at the end of all the instructions, we find this value at $\mathtt{A(N-1)}$. This is important for verification purposes, because then, we need only add the proper boundary constraint in the PIL, referencing a specific step in the execution trace. # commitment-scheme.md: The framework for the proof-verification system of our mFibonacci state machine is that of a polynomial commitment scheme. The mechanism for proving correctness of the computations carried out by the mFibonacci state machine (or, any state machine in the zkEVM setting), is best described in terms of an interactive zero-knowledge proof system. One therefore thinks of the proof-verification system as enabled by an interaction of two parties, traditionally called the verifier and the prover. In practice though, the so-called Fiat-Shamir transformation is used to turn such interactive schemes into non-interactive ones. !!!caution This document dives deep into the technical details of commitment schemes underpinning the zkProver. It describes what a commitment scheme is, the necessary properties such a scheme must possess, and how the previously stated polynomial identities look like in reality. Commitment scheme protocol In the case of our mFibonacci state machine, the prover needs to commit to the polynomials $P(X)$, $Q(X)$, $P(X \omega)$ and $Q(X \omega)$, and the verifier requests the prover to evaluate these polynomials at randomly selected points (i.e., field elements). The general protocol, in an interactive setting, is as follows; 1. The prover commits to a polynomial or a number of polynomials, using a specified polynomial commitment scheme. 2. The verifier randomly selects one or more field elements, which she sends one at a time to the prover as challenges. That is, the verifier requests the prover to provide evaluations (called openings) of the committed polynomials at these randomly selected field elements. 3. This back and forth interaction can occur as many times as the number of openings the verifier deems sufficient to guarantee soundness. 4. The verifier uses relevant polynomial constraints to test veracity of the prover's openings. If all the relevant constraints hold true, then the verifier accepts that the prover has knowledge of the correct polynomials $P(X)$, $Q(X)$, $P(X\omega)$ and $Q(X\omega)$. Properties of commitment schemes For all practical purposes, such the constructed proof-verification system needs to be secure. That is, it must have several cryptographic properties. The most crucial properties are; being hiding and binding, as well as soundness and completeness. - Binding means users are able to commit to values but once committed, it should be impossible for users to change or repudiate their committed values. The committed values are called commitments. - Hiding literally means users can commit to values without revealing the actual values, and it should be infeasible for anyone else to deduce the actual values. - Openness means, given a challenge α , the prover can generate a verifiable proof such that $P(\alpha) = y$. - Soundness has to do with whether it is infeasible for the adversarial prover to convince the verifier to accept invalid proofs (or commitments). A proof system is sound if the probability for the verifier to accept a false proof is less than a third, $\Big(\text{probability} < \dfrac{1}{3} \Big)$. In other words, the soundness property of a proof system requires that any proof created from a false witness should not be convincing to the verifier. - A proof system has completeness if every valid proof is convincing and acceptable to a verifier. Proof systems based on testing polynomial identities take advantage of a basic property of polynomials expressed by the Schwartz-Zippel lemma. According to the Schwartz-Zippel lemma: - For any non-zero polynomial ${Q(X1, \dots , Xn)}$ on ${n}$ variables with degree ${d}$, and ${S}$ a finite but sufficiently large subset of the field $\mathbb{F}$, then values ${ X1, \dots , Xn }$ from ${S}$ are independently and uniformly assigned at random, then ${Pr[Q(X1, \dots , Xn) = 0] â‰¤ \dfrac{d}{|S|}}$. Here's what the Schwartz-Zippel lemma means in the specific case of the mFibonacci state machine: - If the verifier selects the challenges $\{ \alpha1, \alpha2 . . . , \alphal \}$ randomly and uniformly, then the probability of the prover finding a false polynomial ${Q'}$ of degree d, such that ${Q'(\alphaj) = 0 = Q(\alphaj)}$ for all ${j \in \{ 1, 2, \dots , l \}}$, is at most ${\dfrac{d}{|S|}}$, which is very small. This speaks of the soundness of our polynomial commitment scheme. Proving the mFibonacci state machine via PCS Suppose the prover has to prove knowledge of the initial values of the mFibonacci Series that yields the 1024-th term, $A{1023} = \mathtt{14\ 823\ 897\ 298\ 192\ 278\ 947}$. Suppose a certain polynomial commitment scheme (PCS) is used to facilitate proving and verification. In a typical commitment scheme, the following protocol is followed; 1. The prover commits to the polynomials $P(X)$ and $Q(X)$. 2. The verifier selects a random point α, sends it to the prover, with a request for relevant openings. 3. The prover then provides the openings; $P(\alpha)$, $P(\omega \alpha)$, $Q(\alpha)$ and $Q(\omega \alpha)$. 4. The verifier can check correctness of the openings, by using transition constraints, $$ \begin{aligned} \big(1 âˆ' R(X) \big) \big[P(X\cdot \omega) âˆ' Q(X) \big] = \bigg\lvert\mathcal{H}\ 0\qquad\quad\text{ } \\ \big(1 âˆ' R(X) \big) Â· [Q(X\cdot \omega) âˆ' (P(X) Â· Q(X))] = \bigg\lvert\mathcal{H}}\ 0\text{ } \end{aligned} $$ 5. In order to enable the verifier to check the boundary constraint, $$ P(\omega^{\mathtt{1023}}) = \mathtt{14\ 823\ 897\ 298\ 192\ 278\ 947} $$ the prover must send a witness $\large{\mathtt{w}}$ as proof that he or she knows the correct value of the 1024-th term, without disclosing the actual value $$ A{1023} = \mathtt{14\ 823\ 897\ 298\ 192\ 278\ 947}. $$ 6. The verifier then uses a formula, which is specific to the commitment scheme in use and it takes the witness as an input, to check whether the prover has computed the correct $A{1023}$. Note that the prover does not provide any values concerning the constant polynomial $R(X)$, because this is known to both the prover and the verifier. Any PCS such as KZG or FRI-based can be used to efficiently prove correctness of computations of our mFibonacci state machine. Proper range for identities Let's look carefully at the constraints of the mFibonacci state machine; $$ \begin{aligned} \big(1 âˆ' R(X) \big) \cdot \big[P(X\cdot \omega) âˆ' Q(X) \big] = \bigg\lvert\mathcal{H}}\ 0\qquad\quad\text{ }\text{ }\text{ }\text{ } \\ \big(1 âˆ' R(X)\big) Â· [Q(X\cdot \omega) âˆ' (P(X) Â· Q(X))] =

\bigg\lvert{\mathcal{H}}\ 0\text{ }\text{ }\\ \big(P(\omega^{\mathtt{T}}) - \mathcal{K} \big)\cdot R(X) = \bigg\lvert{\mathcal{H}}\ 0\quad\text{ }\text{ }\quad\qquad\quad\quad\qquad\text{ }\text{ }\text{ } \end{aligned} $$ where $\mathtt{T}+1$ is the number of rows in the execution trace and $\mathcal{K}$ is an evaluation of $P(X)$ at $\omega^{\mathtt{T}}$, corresponding to the value of the registry $\mathtt{A}$ in the $(\mathtt{T}+1)$-st state, when specific initial values $\mathtt{A}0$ and $\mathtt{B}0$ are used in the state machine. The problem with the transition constraints, as presented above, is that they hold true only for $X \in \mathcal{H}$, and not necessarily for $X \in \mathbb{F}p$. Note that the left-hand sides of the polynomial identities are but polynomials, which can be labelled $p1(X)$, $p2(X)$ and $p3(X)$. That is; $$ \begin{aligned} p1(X) =\big(1 â^' R(X) \big) \cdot \big[P(X\cdot \omega) â^' Q(X) \big] = \bigg\lvert{\mathcal{H}}\ 0\qquad\quad\text{ }\text{ }\text{ }\text{ } \\ p2(X) = \big(1 â^' R(X)\big) Â· [Q(X\cdot \omega) â^' (P(X) Â· Q(X))] = \bigg\lvert{\mathcal{H}}\ 0\text{ }\text{ }\\ p3(X) = \big(P(\omega^{\mathtt{T}}) - \mathcal{K} \big)\cdot R(X) = \bigg\lvert{\mathcal{H}}\ 0\quad\text{ }\text{ }\text{}\qquad\qquad\qquad\qquad \end{aligned} $$ Define the so-called vanishing polynomial on $\mathcal{H}$, denoted by $\mathtt{Z}{\mathcal{H}}(X)$, as the monomial of maximal degree such that $\mathtt{Z}{\mathcal{H}}(X) = 0$ for all $X \in \mathcal{H}$. Therefore; $$ \mathtt{Z}{\mathcal{H}}(X) = (X-1)\cdot(X-\omega)\cdot(X-\omega^2)\cdots(X-\omega^{n-1}) = X^{n} - 1 $$ Since $pi(X) = 0$ for all $X \in \mathcal{H} = \{ \omega, \omega^2, \omega^3, \dots , \omega^n = 1 \}$, then; $$ pi(X) =\ \big((X-1)\cdot(X-\omega)\cdot(X-\omega^2)\cdots(X-\omega^{n-1})\big)\cdot qi(X)\ =\ \big(X^{n} - 1 \big) \cdot qi(X) $$ for some quotient polynomial $qi(X)$, for each $i \in \{ 1, 2, 3 \}$. The polynomial identities of our mFibonacci state machine can therefore be rewritten as: $$ \big(1 â^' R(X) \big) \cdot \big[P(X\cdot \omega) â^' Q(X) \big] = \mathtt{Z}{\mathcal{H}}(X)\cdot q1(X) \qquad\quad\text{ }\text{ }\text{ }\text{ } \\ $$ $$ \big(1 â^' R(X)\big) Â· [Q(X\cdot \omega) â^' (P(X) Â· Q(X))] = \mathtt{Z}{\mathcal{H}}(X)\cdot q2(X) \text{ }\text{ }\\ $$ $$ \big(P(\omega^{\mathtt{T}}) - \mathcal{K} \big)\cdot R(X) = \mathtt{Z}{\mathcal{H}}(X)\cdot q3(X) \qquad\text{ }\text{}\qquad\qquad\quad\qquad\text{ } $$ The representatives $R(X)$ and $Z{\mathcal{H}}(X)$ in the PCS, can be preprocessed and be made public (i.e., known to both the prover and the verifier). The verifier can check specific openings of these polynomials, $R(X)$ and $Z{\mathcal{H}}(X)$. # mfibonacci-example.md: Consider a proof-verification scheme, using an arbitrary Polynomial Commitment Scheme. In this scheme, users must prove knowledge of the Nth member of a multiplicative Fibonacci series for specific initial conditions. What is a multiplicative Fibonacci series? The multiplicative Fibonacci series (or simply mFibonacci series), denoted by $\mathbf{a0, a1, a2, \dots , an}$ has the property that the product of every two consecutive members $\mathbf{a{i-1}}$ and $\mathbf{ai}$ gives the value of the next member $\mathbf{a{i+1}}$. That is, $\mathbf{ a{i+1} = a{i-1}\cdot ai }$. Also, the initial values are specified as $\mathbf{a0} = 2$ and $\mathbf{a1} = 1$. Here are the first ten terms of the mFibonacci series, $$ \mathbf{ \ 2,\ 1,\ 2,\ 2,\ 4,\ 8,\ 32,\ 256,\ 8192,\ 2097152,\ \dots } $$ As a trivial example, the challenge may be: Prove knowledge of the initial values that produced $\mathbf{a{10} = 17179869184}$, the eleventh member of the mFibonacci series, without revealing the initial values. The task therefore, is to first build a state machine that would enable anyone to prove knowledge of the initial values $\mathbf{a0}$ and $\mathbf{a1}$ that yields a specific N-th member of the mFibonacci series. Constructing mFibonacci state machine Consider a state machine with two registries $\mathbf{A}$ and $\mathbf{B}$ where $$ \begin{aligned} &\mathbf{A} = [A0, A1, \dots , AT] \\ &\mathbf{B} = [B0, B 1, \dots , BT] \end{aligned} $$ such that the i-th state is the pair $\big(Ai , Bi \big)$. Such a state machine is an mFibonacci state machine if indeed the registry values conform to the format of the mFibonnacci series. See Figure 4 below, for an mFibonacci state machine with the initial conditions, $A0 = 2$ and $B0 = 1$. !Figure 4: mFibonacci SM with two registries The state transitions from $\mathtt{S} = \big(Ai , Bi \big)$ to $\mathtt{S}' = \big(A{i+1} , B{i+1} \big)$ conform to the following constraints; $$ \begin{aligned} { A{i+1} = Bi \quad\text{ }\text{ }\text{ } } \\ { B{i+1} = Ai \cdot Bi } \end{aligned} $$ The aim here is to; express the evolution of the execution trace in terms of polynomials, build corresponding polynomial identities, and ultimately construct a ZK proof/verification scheme for our mFibonacci state machine. Building polynomial identities The polynomials that represent the two registries are taken from the set of polynomials $\mathbb{F}p [X]$, where the coefficients are elements of a prime field $\mathbb{F}p$ and $p = 2^{64} â^' 2^{32} + 1$. The polynomials are evaluated over the subgroup $$ {\mathcal{H}} = \{ \omega, \omega^2, \omega^3, \dots , \omega^7, \omega^8 = 1 = \omega^0 \} = \langle \omega \rangle \subseteq \mathbb{F}p^ $$ of order 8. Define two polynomials $P(X)$ and $Q(X)$ such that: $$ \begin{aligned} P(\omega^i) = A[i] \ \ \iff \ \ A = [2,1,2,2,4,8,32,256]\quad\text{ }\text{ } \\ Q(\omega^i) = B[i] \ \ \iff \ \ B = [1,2,2,4,8,32,256,8192] \end{aligned} $$ Since every X in ${\mathcal{H}}$ is of the form $X = \omega^i$ for some i, we have $$ \begin{aligned} P(X\cdot \omega) &= P(\omega^{i + 1}) = A{i+1}, \\ Q(X\cdot \omega) &= Q(\omega^{i+1}) = B{i+1}. \end{aligned} $$ The previously stated constraints, imposed on the state transitions $\mathtt{S} \to \mathtt{S}'$ of the mFibonacci state machine, translate into the following polynomial identities; $$ P(X\cdot \omega) = \bigg\lvert{\mathcal{H}}\ Q(X),\qquad\text{ } \\ Q(X\cdot \omega) = \bigg\lvert{\mathcal{H}}\ P(X) \cdot Q(X) $$ If these polynomial identities should accurately express the two registries, then every state transition of the mFibonacci SM must satisfy them. Non-cyclicity of the mFibonacci state machine Note that the definition of ${\mathcal{H}}$ does not restrict the values of i to be less than 8. Even if we set $i = 27$, the element ω^{27} is in ${\mathcal{H}}$ because $$ \begin{aligned} \omega^{27} = w^8 \cdot \omega^8 \cdot \omega^8 \cdot \omega^3 = 1 \cdot 1 \cdot 1 \cdot \omega^3 = \omega^3 \end{aligned} $$ However, the unrestricted value of i, which implies there is no bound on the number of state changes (i.e., on the clock), presents problems with the above polynomial identities. Let us test if the polynomial identities hold true for all permissible values of i. So let $X = \omega^7$ and refer to the registry values given in Figure 4. - For the first identity we get, $$ \begin{aligned} P(X\cdot \omega) = P(\omega^7 \cdot \omega) = P(\omega^8) = P(\omega^0) = A0 = 2\ \ \ \text{ but }\ Q(X) = Q(\omega^7) = 8192 \not= 2 \quad\qquad\qquad\qquad\quad \end{aligned} $$ - Similarly, for the second identity, we get, $$ \begin{aligned} Q(X\cdot \omega) = Q(\omega^7 \cdot \omega) = Q(\omega^8) = Q(\omega^0) = B0 = 1\ \ \text{ but } \qquad \\ P(X)\cdot Q(X) = P(\omega^7)\cdot Q(\omega^7) = 256\cdot 8192 = 2097152 \not= 1 \quad \end{aligned} $$ Clearly, the polynomial identities are not aligned with the registry values of the mFibonacci SM. The reason for this disparity is that, whereas H is cyclic, the polynomial identities are not. Introducing cyclicity In order to inject some cyclicity into the mFibonacci SM, we add a third registry $\mathbf{C} = [C1, C2, \dots , CT]$ and set the registry values to $\mathbf{C} = [0, 0, \dots , 0, 1]$. Hence the mFibonacci SM is as depicted in Figure 5 below. !mFibonacci SM with three registries The corresponding polynomial $R(x)$ is defined as follows; $$ R(\omega^i) = C[i] $$ That is; $$ \begin{aligned} R(\omega^i) &= C[i] = 1, \text{ if } \ \ (i+1)\mod 8 = 0 \\ R(\omega^i) &= C[i] = 0, \text{ otherwise} \end{aligned} $$ The polynomial $R(x)$ is incorporated into the previous polynomial identities as follows; $$ \begin{aligned} P(X \cdot \omega) = \bigg\lvert{\mathcal{H}}\ \ Q(X) \cdot \big(1 â^' R(X) \big) + R(X)\cdot A0 \quad\quad\text{ }\text{ }\text{ } \\ Q(X\cdot \omega) = \bigg\lvert{\mathcal{H}}\ \ (1 â^' R(X)) \cdot P(X)\cdot Q(X) + R(X)\cdot B0 \end{aligned} $$ Note that, for all the states where the new registry $C[i] = 0$, these new identities coincide with the previous ones (where only registries A and B were used). These polynomial identities can be rewritten as; $$ \begin{aligned} \big(1 â^' R(X) \big) \cdot \big[P(X\cdot \omega) â^' Q(X) \big] + R(X)\cdot \big[P(X \cdot \omega) â^' A0 \big] = \bigg\lvert{\mathcal{H}}\ 0\quad\text{ }\text{ }\text{ } \\ (1 â^' R(X)) Â· [Q(XÏ‰) â^' (P(X) Â· Q(X))] + R(X)\cdot [Q(XÏ‰) â^' B0] = \bigg\lvert{\mathcal{H}}\ 0 \end{aligned} $$ Let's check if these identities are cyclic. Again, let $X = \omega^7$ and use the registry values given in the figure. - For the first identity, we observe that, $$ \begin{aligned} \text{LHS} &= P(X\cdot \omega) = P(\omega^7 \cdot \omega) = P(\omega^8) = P(\omega^0) = A0 = 2\ \ \text{ and } \\ \text{RHS} &= Q(X)\cdot \big(1 â^' R(X) \big) + R(X)\cdot A0\ =\ Q(\omega^7)\cdot \big(1 â^' 1 \big) + 1\cdot 2 = 2 = \text{LHS} \end{aligned} $$ - Similarly, for the second identity, we observe that, $$ \begin{aligned} \text{LHS} &= Q(X\cdot \omega) = Q(\omega^7 \cdot \omega) = Q(\omega^8) = Q(\omega^0) = B0 = 1 \ \ \text{ and } \text{ and } \\ \text{RHS} &= (1 â^' R(X)) \cdot P(X)\cdot Q(X) + R(X)\cdot B0 \\ &= (1 â^' 1) \cdot P(\omega^7)\cdot Q(\omega^7) + 1\cdot 1 = 0 + 1 = \text{LHS} \end{aligned} $$ These polynomial identities enforce correct state transitioning, and are therefore referred to as transition constraints. They apply to every pair of consecutive states. That is, every pair of consecutive rows in the execution trace of the SM. Verifying computations In addition to transition constraints, are boundary constraints. A boundary constraint is a constraint that enforces that a polynomial has a certain value at a particular root of unity. Varied initial conditions Note that instead of being restricted to the given initial conditions $\big(A0, B0\big) = \big(2, 1\big)$ the mFibonacci state machine together with its polynomial identities can be adjusted to any initial conditions $\big(A0, B0\big)$. For example, for $A0 = 23$ and $B0 = 46$, the constraints should be; $$ \begin{aligned} \big(1 â^' R(X) \big) \cdot \big[P(X\cdot \omega) â^' Q(X) \big] + R(X)\cdot \big[P(X \cdot \omega) â^' 23 \big] = \bigg\lvert{\mathcal{H}}\ 0\quad\text{ }\text{ }\text{ } \\ (1 â^' R(X)) Â· [Q(XÏ‰) â^' (P(X) Â· Q(X))] + R(X)\cdot [Q(XÏ‰) â^' 46] = \bigg\lvert{\mathcal{H}}\ 0 \end{aligned} $$ In the context of our mFibonacci SM, the verifier can set the initial conditions $\big(A0 , B0 \big)$ to values of his or her own choice, and generate the state machine while keeping $A0$ and $B0$ secret. The prover's task is therefore, to prove knowledge of $A0$ and $B0$ that led to a given N-th term of the mFibonacci series. Boundary constraints Boundary constraints apply to particular registry values, and are used to enforce that the correct initial state was applied. The idea here is to set up a specific boundary constraint, which the verifier can use to check that correct initial conditions were applied, when the prover was computing a particular Nth term of the mFibonacci series. Yet, the verifier must not disclose any information about the secret values $A0$ and $B0$. Therefore, the first thing to do, is removing terms in the identities bearing the initial values $A0$ and $B0$. This means modifying our polynomial identities to the ones below; $$ \begin{aligned} \big(1 â^' R(X) \big) \cdot \big[P(X\cdot \omega) â^' Q(X) \big] = \bigg\lvert{\mathcal{H}}\ 0\qquad\text{ }\text{ }\text{ } \\ (1 â^' R(X)) Â· [Q(X\cdot \omega) â^' (P(X) Â· Q(X))] = \bigg\lvert{\mathcal{H}}\ 0 \end{aligned} $$ Secondly, knowing that $A0$ and $B0$ yield the k-th term $A{k-1} = P(\omega^{k-1}) =: \mathcal{K}$, the verifier adds the boundary constraint; $$ P(\omega^{k-1}) = \mathcal{K} $$ In other words, the prover has to provide three polynomials $P(X)$, $Q(X)$, $P(X\omega)$ and $Q(X\omega)$ together with the correct k-th term. The verifier then tests if these polynomials conform to the above constraints. If all three constraints are satisfied, then the prover knows the correct initial values $A0$ and $B0$. This logic is valid simply because the computations carried out by the state machine are deterministic by nature. Proof elements and verification All computations are carried out in a field $\mathbb{F}p$, where $p = \mathtt{2^{64}-2^{32}+1}$, a Goldilocks-like prime number. Suppose the verifier knows that an mFibonacci series starting with initial values, $A0$ and $B0$, yields $A{\mathtt{1023}} = \mathtt{14\ 823\ 897\ 298\ 192\ 278\ 947}$ as the value of the $\mathtt{1024}$-th term. The verifier can challenge anyone to prove knowledge of the initial condition of the mFibonacci SM to provide three polynomials and the correct $\mathtt{1024}$-th term. That is, the verifier uses the following constraints to verify the prover's submissions: $$ \begin{aligned} \big(1 â^' R(X) \big) \cdot \big[P(X\cdot \omega) â^' Q(X) \big] = \bigg\lvertH\ 0\qquad\quad\text{ }\text{ }\text{ } \\ \big(1 â^' R(X)\big) Â· [Q(X\cdot \omega) â^' (P(X) Â· Q(X))] = \bigg\lvertH\ 0 \\ \big(P(\omega^{\mathtt{1023}}) - \mathcal{K} \big)\cdot R(X) = 0\qquad\quad\qquad\quad\text{ } \end{aligned} $$ Anyone who knows the three polynomials and the correct initial conditions, say $A0 = 234$ and $B0 = 135$, can simply run the mFibonacci SM code to compute $A{\mathtt{1023}} = P(\omega^{\mathtt{1023}})$. See below figure for the JS code. !Code Example of the mFibonacci SM's Computation Trace # mfibonacci.md: Introduction This document gives a short summary of the multiplicative Fibonacci state machine, presented as a simple model for the zkProver state machines. As already mentioned in preceding sections, the overall design of the Polygon zkEVM follows the state machine model, and thus emulates the Ethereum virtual machine (EVM). The computations involved in Ethereum such as, making payments, transferring ERC20 tokens and running smart contracts, are repeatedly carried out and are all deterministic. That is, a particular input always produces the same output. Unlike the arithmetic circuit model which would need loops to be unrolled and hence resulting in undesirably larger circuits, the state machine model is most suitable for iterative and deterministic computations. !Deterministic computation zkProver's state machine design To break down the complexity of the zkProver's design, we use a simplified Hello World example. In particular, the multiplicative Fibonacci state machine. This simple state machine helps illustrate, in a general sense, how the state machine approach has been implemented to realize the zkProver. !!!info Computing consecutive members of the well-known Fibonacci series, starting with specific initial values, is a deterministic computation. Consider a scenario where a party called the prover, needs to prove knowledge of the initial values of the Fibonacci series used to produce a given N-th value of the series, in a verifiable manner. These computations serve as an ideal analogy for the zkProver's role, which is to generate verifiable proofs confirming the validity of transactions submitted to the Ethereum blockchain. The approach involves developing a state machine that enables a prover to generate and submit a verifiable proof of knowledge, which anyone can then use to verify its validity. !A Skeletal View of the Design Process The process that leads to achieving such a state machine-based system takes a few steps: - Modeling deterministic computation as a state machine. - Specifying the equations that fully describe the state transitions of the state machine, known as arithmetic constraints. - Using established and efficient Mathematical methods to define the corresponding polynomials. - Expressing the previously stated arithmetic constraints into their equivalent polynomial identities. These polynomial identities are equations that can be easily tested in order to verify the prover's claims. A so-called commitment scheme is required for facilitating the proving and the verification. Hence, in the zkProver context, a proof-verification scheme called PIL-STARK is used. This series of documents on the multiplicative Fibonacci state machine culminates in a DIY guide to implementing the proving and verification of the mFibonacci state machine. # pil-stark-demo.md: !!!note This document is a DIY implementation guide for the PIL-STARK proof-verification system. Before delving into the implementation, we first ensure the boundary constraints in the $\texttt{mFibonacci.pil}$ code are not hardcoded. The aim here is to implement these values as $\texttt{publics}$. This subsequently populates the $\texttt{publics}$' field in the parsed $\texttt{\{ \} fibonacci.pil.json}$ file. Boundary constraints as publics As seen in the description of PIL-STARK, one of the outputs of the SM-prover is the $\texttt{publics}$, which are publicly known values. However, looking back at our demonstration, where the $\texttt{mFibonacci.pil}$ file was compiled with $\texttt{PILCOM}$, you can notice that the $\texttt{publics}$ field in the parsed $\texttt{\{ \} fibonacci.pil.json}$ file is empty. Check line 6 in the provided snapshot of the "$\texttt{\{ \} fibonacci.pil.json}$" file, and notice it reads thus: $$ \texttt{"publics": [],} $$ This is so, because no $\texttt{publics}$ were defined in the $\texttt{mFibonacci.pil}$ file that we compiled. Again, you'll notice that the 1024-th value of the polynomial $\texttt{a}$ was hardcoded as $\mathtt{14411518804230144 0}$ in $\texttt{line}$ $\texttt{13}$, which checks the boundary constraint: $$ \mathtt{ISLAST(a - 14411518804230144 0) = 0;} $$ This is undesirable because any change in the initial conditions would yield a wrong proof, which the verifier would reject. The aim in this subsection is to therefore, rewrite the boundary constraint in the $\texttt{mFibonacci.pil}$ file such that, instead of hardcoding this public value, it is rather generically defined. So, $\mathtt{14411518804230144 0}$, is replaced with a public variable called "$\texttt{out}$". The new variable must be declared, and it is defined as; $$ \textbf{public} \texttt{ out = a(\%N - 1);} $$ The modified boundary constraint check in $\texttt{mFibonacci.pil}$ now looks like this; $$ \texttt{ISLAST(a - :out) = 0;} $$ where the : colon-prefix indicates a read of the value stored at $\texttt{out}$. The modified $\texttt{mFibonacci.pil}$ file, before compilation with $\texttt{PILCOM}$, is now as follows, !mFibonacci.pil file with "publics" This modified $\texttt{mFibonacci.pil}$ file can be compiled with $\texttt{PILCOM}$ in the manner demonstrated earlier. The resulting parsed PIL file, "$\texttt{\{ \} mfibonacci.pil.json}$", now reflects some information in the "$\texttt{publics}$" field, as shown below. !A non-empty "publics" field the parsed PIL file PIL-STARK implementation guide Here is a step-by-step guide on how PIL-STARK is implemented. You are encouraged to Do-It-Yourself by following the steps provided below. Initialise a node project The first step is to make a new subdirectory named mfibonaccism, switch directory to it and initialise a node project as indicated below; bash npm init -y A successful initialisation looks like this: !Successful node initialisation Next, install the required dependencies with the following command, bash npm install pil-stark yargs chai The installation takes seconds, and again the results looks like this, !Installed dependencies Create input files First of all, the overall inputs to PIL-STARK are; the $\texttt{.pil}$ file describing the mFibonacci state machine, the STARK configuration $\texttt{.json}$ file and the inputs $\texttt{.json}$ file, which contains initial values of the mFibonacci state machine. - Either move the modified version of the $\texttt{mfibonacci.pil}$ file to the $\mathtt{mfibonacci\sm}$ subdirectory, or create a new file afresh at this subdirectory. Call it $\texttt{mfibonacci.pil}$. That is, copy the text below to the new file. Save the file. pil title="mfibonacci.pil" constant %N = 1024; namespace mFibonacci(%N); pol constant ISLAST; // 0,0,0,0,.....,1 pol commit a, b; pol ab = ab; // publics public out = a(%N-1); // transition constraints (1-ISLAST) (a' - b) = 0; (1-ISLAST) (b' - (ab)) = 0; // boundary constraint ISLAST(a-:out)=0; - Next, create a new file and call it mfib.starkstruct.json. Copy the code provided below into this JSON file, and save it. json { "nBits": 10, "nBitsExt": 11, "nQueries": 8, "verificationHashType": "GL", "steps": [{"nBits": 11}, {"nBits": 7}, {"nBits": 3}] } - Create a new file and call it mfib.input.json. Populate this JSON file with the initial values of the mFibonacci state machine of your choice (the numbers must be positive integers). We use here $\texttt{[234, 135]}$. Simply type the 2-element array in the mfib.input.json file and save it. Create the executor In our earlier description of PIL-STARK, the $\texttt{executor}$ was 'split' into the setup executor and the prover executor. This was done for the sake of simplicity. The two $\texttt{executors}$ are but one program that generates the evaluations of the constant polynomial, as well as the evaluations of the committed polynomials. Create a new file and call it executormfibonacci.js. Copy the code-text shown below, into this .js file and save it in the mfibonaccism subdirectory. js title="executormfibonacci.js" const { FGL } = require("pil-stark"); module.exports.buildConstants = async function (pols) { const N = 1024; for (let i=0; i 2$. In which case, the $\mathbf{(l + j)}$-th extension branch $\mathbf{B{ext(l + j)}}$, with the $\mathbf{L{new}}$ and $\mathbf{L{z}}$ as its child-nodes, completes the create operation. An update of values The create operation is actually only complete once all the values on the navigation path from the new root to the new leaf are updated. Example: A create operation with single branch extension Suppose a leaf needs to be created to store a new key-value pair $\big({K{\mathbf{new}}\ } , V\mathbf{{new}}\big)$, where $K{\mathbf{new}} = 11010110$. Consider the SMT shown in the below figure. !create operation - Non-zero Leaf Node In this example, navigation using the least-significant key-bits, $\text{kb}\mathbf{0} = 0$ and $\text{kb}\mathbf{1} = 1$, leads to an existing leaf $\mathbf{L{\mathbf{c}}}$. And the key-value pair $(V\mathbf{\mathbf{c}}, \text{HV}\mathbf{\mathbf{c}})$ stored at $\mathbf{L{\mathbf{c}}}$ has the key $K{\mathbf{c}} = 11010010$. 1. Value-inclusion check. A value-inclusion check of $V\mathbf{\mathbf{c}}$ is performed before creating any new leaf. Since this amounts to a read operation, which has been illustrated in previous examples, we omit how this is done here. Once $V\mathbf{\mathbf{c}}$ passes the check, the create operation continues by inserting the new leaf. 2. New leaf insertion. In this example, the new leaf $\mathbf{L{new}}$ cannot be inserted at the key-address 01 where $\mathbf{L{\mathbf{c}}}$ is positioned. A branch extension $\mathbf{{B}{ext}}$ must therefore be done at the address 01 with the leaves $\mathbf{L{new}}$ and $\mathbf{L{c}}$ as child-nodes. Since the third least-significant key-bits of $K{\mathbf{new}}$ and $K{\mathbf{c}}$ are not the same, $\text{kb}\mathbf{2new} = 1$ and $\text{kb}\mathbf{2c} = 0$, the addresses 110 and 010 of the leaves $\mathbf{L{new}}$ and $\mathbf{L{c}}$, respectively, are distinct. Therefore, no further extension is necessary. And the create operation is complete by updating all the values on the navigation path. The next process, after forming the branch extension, is to update all the nodes along the path from the root to the new leaf $\mathbf{L{new}}$. The verifier follows the steps of the update operation to accomplish this. 3. An update of SMT values. The verifier computes the following, 1. The hashed value, $\text{HV}{\mathbf{new}} = \mathbf{H{noleaf}} (V\mathbf{new})$. 2. The new leaf, $\mathbf{L{new}} = \mathbf{H{leaf}} (\text{RK}{\mathbf{new}} \| \text{HV}{\mathbf{new}})$. 3. Then, $\mathbf{B{ext}} = \mathbf{H{noleaf}} (\mathbf{L{c}} \| \mathbf{L{new}})$. 4. Again, $\mathbf{B{new}} = \mathbf{H{noleaf}}(\mathbf{S{ab}} \| \mathbf{B{ext}})$. 5. And finally computes, $\mathbf{{root}{new}} = \mathbf{H{noleaf}}(\mathbf{B{new}} \| \mathbf{L{d}})$. This illustrates how the create operation is performed at a non-zero leaf, when only one branch extension is required. Example: A create operation with multiple branch extensions This example provides an illustration of the create operation at a non-zero leaf, where more than one branch extensions are required. Suppose a leaf must be created to store a new key-value pair $\big(K{\mathbf{new}}, V\mathbf{new}\big)$, where $K{\mathbf{new}} = 11010110$. Consider the SMT shown in the below figure. !create operation - Three Branch Extensions Navigating the tree by using the least-significant key-bits, $\text{kb}\mathbf{0} = 0$ and $\text{kb}\mathbf{1} = 1$, leads to an existing leaf $\mathbf{L{\mathbf{c}}}$. In this example, suppose the key-value pair $(K{\mathbf{c}}, \text{HV}\mathbf{\mathbf{c}})$ stored at $\mathbf{L{\mathbf{c}}}$ has the key $K{\mathbf{c}} = 11100110$. 1. Value-inclusion check. Before creating the new leaf, it is important to first check if $V\mathbf{\mathbf{c}}$ is indeed included in the root, $\mathbf{root}\mathbf{abcd}$. Since this amounts to performing a read operation, which has been illustrated in previous examples, we omit here how this is done. Once $V\mathbf{\mathbf{c}}$ passes the value-inclusion check, the create operation proceeds with inserting the new leaf. 2. New leaf insertion. Note that the first and second least-significant key-bits for both $K\mathbf{new}$ and $K\mathbf{c}$ are the same. That is, $\text{kb}\mathbf{0new} = 0 = \text{kb}\mathbf{0c}$ and $\text{kb}\mathbf{1new} = 1 = \text{kb}\mathbf{1c}$. As a result, the new leaf $\mathbf{L{new}}$ cannot be inserted at the key-address 01, where $\mathbf{L{\mathbf{c}}}$ is positioned. An extension branch $\mathbf{{B}{ext1}}$ is formed at the tree-address 01. But, can the leaves $\mathbf{L{new}}$ and $\mathbf{L{c}}$ be child-nodes to $\mathbf{{B}{ext1}}$? Since the third least-significant key-bits of $K\mathbf{new}$ and $K\mathbf{c}$ are the same; that is, $\text{kb}\mathbf{2new} = 1 = \text{kb}\mathbf{2c}$; leaves $\mathbf{L{new}}$ and $\mathbf{L{c}}$ cannot be child-nodes to $\mathbf{{B}{ext1}}$. Another extension branch $\mathbf{{B}{ext2}}$ is formed at the tree-address 011. Again, since the fourth least-significant key-bits of $K\mathbf{new}$ and $K\mathbf{c}$ are the same; $\text{kb}\mathbf{3new} = 0 = \text{kb}\mathbf{3c}$; the leaves $\mathbf{L{new}}$ and $\mathbf{L{c}}$ cannot be child-nodes to $\mathbf{{B}{ext2}}$. A third extension branch $\mathbf{{B}{ext3}}$ is needed at the tree-address 0110. In this case, the fifth least-significant key-bits of $K\mathbf{new}$ and $K\mathbf{c}$ are different, i.e., $\text{kb}\mathbf{4new} = 1$ and $\text{kb}\mathbf{4c} = 0$. The leaves $\mathbf{L{new}}$ and $\mathbf{L{c}}$ are now made child-nodes of the extension branch $\mathbf{{B}{ext3}}$. See the above figure. Once unique addresses for the key-value pairs $\big(K\mathbf{{c}} , V\mathbf{c} \big)$ and $\big(K\mathbf{{new}} , V\mathbf{{new}}\big)$ are reached, and the leaf $\mathbf{L{new}}$ is inserted, all the nodes along the navigation path from the new leaf $\mathbf{L{new}}$ to the root are updated as follows. 3. An update of SMT values. The verifier computes, 1. The hash of the new value, $\text{HV}\mathbf{new} = \mathbf{H{noleaf}}(V\mathbf{new})$. 2. The new leaf, $\mathbf{L{new}} = \mathbf{H{leaf}} (\text{RK}\mathbf{new} \| \text{HV}\mathbf{new}})$. 3. Then, $\mathbf{B{ext3}} = \mathbf{H{noleaf}}(\mathbf{L{c} \| \mathbf{L{new}})$. 4. Followed by $\mathbf{B{ext2}} = \mathbf{H{noleaf}}(\mathbf{B{ext3}} \| \mathbf{0})$. 5. And, $\mathbf{B{ext1}} = \mathbf{H{noleaf}}(\mathbf{0} \| \mathbf{B{ext2}})$. 6. Again, $\mathbf{B{new}} = \mathbf{H{noleaf}}(\mathbf{S{ab}} \| \mathbf{B{ext2}})$. 7. Finally computes, $\mathbf{{root}{new}} = \mathbf{H{noleaf}}(\mathbf{B{new}} \| \mathbf{L{d}})$. This completes the create operation at an existing leaf where several branch extensions are needed. Note that the create operation at a non-leaf node clearly changes the topology of the tree. The delete operation The delete operation refers to a removal of a certain key-value pair from a binary SMT. It is in fact the reverse of the create operation. There are two types of scenarios that can occur when executing a delete operation. There is a scenario where a delete operation is equivalent to an update operation of a non-zero leaf to a NULL leaf. In this case the topology of the SMT does not change. This occurs when the leaf being deleted has a non-zero sibling-node. On the other hand, a delete operation can be tantamount to the reverse of a create operation where extension branches are removed from the tree. The topology of the SMT can drastically change. This scenario occurs when the leaf being removed has a zero sibling-node. A delete operation involves two main steps; 1. A read of the value to be deleted is executed. That is, - Navigating to the value. - Checking if the value is included in the root. - Checking if the given key (reconstructed from the given Remaining Key and the least-significant key-bits) matches the key at the leaf (reconstructed from the Remaining Key found at the leaf and the given key-bits). 2. This step depends on whether the sibling of the leaf to be deleted is zero or not; - If the sibling is not a zero node, an update to a zero is performed. - If the sibling is a zero-node, an update to a zero is performed and the parent-node is turned into a NULL node with no child-nodes. Delete leaves with non-zero siblings Consider a delete of a key-value pair $\big(K{\mathbf{b}} , V\mathbf{b} \big)$ where its leaf $\mathbf{Lb}$ has a non-zero node sibling. Suppose the data provided includes; the Remaining Key $\tilde{\text{RK}}\mathbf{b}}$, the least-significant key-bits $\text{kb}0 = 0$ and $\text{kb}1 = 1$, the root $\mathbf{{root}{abc}}$, and the sibling $\mathbf{La}$ which is not a zero node and the leaf $\mathbf{Lc}$. With reference to the figure below, navigation leads to the leaf $\mathbf{Lb}$. !delete operation - Non-Zero Sibling Next, perform a Merkle proof to check if the hashed value $\text{HV}\mathbf{b}$ at $\mathbf{Lb}$ is included in the given root; - Compute $\tilde{\mathbf{L}}\mathbf{b} = \mathbf{H{leaf}} (\text{RK}\mathbf{b}} \| \text{HV}\mathbf{b})$. - Then $\tilde{\mathbf{B}}\mathbf{ab} = \mathbf{H{noleaf}} (\mathbf{La} \| \tilde{\mathbf{L}}\mathbf{b})$. - And, $\tilde{\mathbf{root}}\mathbf{abc} = \mathbf{H{noleaf}} (\tilde{\mathbf{B}}\mathbf{ab} \| \mathbf{Lc})$. - Check if $\tilde{\mathbf{root}}\mathbf{abc}$ equals $\mathbf{{root}{abc}}$. Simultaneously, check if $\tilde{K}{\mathbf{b}}$ equals $\text{K}{\mathbf{b}}$, where $\tilde{K}{\mathbf{b}} = \tilde{\text{RK}}{\mathbf{b}} \| \text{kb}1 \| \text{kb}0$ and $\text{K}{\mathbf{b}} = \text{RK}{\mathbf{b}} \| \text{kb}1 \| \text{kb}0$ are keys reconstructed while climbing the tree. Since the sibling $\mathbf{La}$ is not a zero node, the hashed value $\text{HV}\mathbf{b}$ found at the leaf $\mathbf{Lb}$ is updated to a zero. And the values along the navigation path are also updated accordingly. That is, - The leaf $\mathbf{Lb}$ is set to "$\mathbf{0}$", a zero node. - The parent-node is now, $\mathbf{B{a0}} = \mathbf{H{noleaf}} (\mathbf{La} \| \mathbf{0})$. - And, the new root, $\mathbf{{root}{abc}} = \mathbf{H{noleaf}}(\mathbf{B{a0}} \| \mathbf{La})$. Notice how the SMT maintains its original shape. Delete leaves with zero siblings Consider deleting a key-value pair $\big(K{\mathbf{c}} , V\mathbf{c} \big)$ where its leaf $\mathbf{Lc}$ has a zero-node sibling. Suppose the data provided includes; the Remaining Key $\tilde{\text{RK}}{\mathbf{c}}$, the least-significant key-bits $\text{kb}0 = 0$, $\text{kb}1 = 1$ and $\text{kb}2 = 1$, the root $\mathbf{{root}{a0cd}}$, and the sibling "$\mathbf{0}$" which is a zero node, and the leaves $\mathbf{La}$ and $\mathbf{Ld}$. With reference to the figure below, navigation leads to the leaf $\mathbf{Lc}$. !Figure 14(../../../img/zkEVM/fig14-b-dlt-z-sib.png): delete operation - Zero Sibling The read step in this case is similar to what is seen in the above case. The update step depends on the sibling of $\mathbf{Lc}$. Since the sibling is $\mathbf{0}$, an update of $\mathbf{Lc}$ to zero results in the branch $\mathbf{B{0c}}$ having two zero nodes as child-nodes. Since $\mathbf{H{noleaf}} (\mathbf{0} \| \mathbf{0}) = 0$, it is therefore expedient to turn the branch $\mathbf{B{0c}}$ into a zero node with no child-nodes. That is, the update step of this delete operation concludes as follows; - The original branch $\mathbf{B{0c}}$ is now "$\mathbf{0}$", a zero node. - The parent-node is now, $\mathbf{B{a0}} = \mathbf{H{noleaf}} (\mathbf{La} \| \mathbf{0})$. - And, the new root, $\mathbf{{root}{a0d}} = \mathbf{H{noleaf}}(\mathbf{B{a0}} \| \mathbf{Ld})$. Notice that in this example, the delete operation alters the topology of the SMT. Conclusion The operations discussed in this document are in fact the very actions the main state machine instructs the storage state machine to perform. The prover and the verifier, as used in the above explanations, can loosely be interpreted as the executor of the storage state machine and the storage SM's PIL code, respectively. The zero-knowledge Assembly (zkASM) of the storage state machine plays the facilitator's role. The zkASM is the interpreter between the storage state machine and the main state machine, also between the storage state machine and the Poseidon state machine. The two hash functions used in building the storage binary SMTs, are special versions of the Poseidon family of hash functions. # detailed-smt.md: This document covers more concepts needed in understanding our specific design of the zkProver storage using binary SMTs. These concepts also help in elucidating how keys influence the shape of binary SMTs. First is the level of a leaf. The level of a leaf, $\mathbf{L}{\mathbf{x}}$, in a binary SMT is defined as the number of edges one traverses when navigating from the root to the leaf. Denote the level of the leaf $\mathbf{Lx}$ by $\text{lvl}(\mathbf{Lx})$. Leaf levels Consider the below figure for an SMT storing seven key-value pairs, built by following the principles explained in the foregoing subsection; $$ (\mathbf{K}{\mathbf{a}}, V{\mathbf{a}}),\ (\mathbf{K}{\mathbf{b}}, V{\mathbf{b}}),\ (\mathbf{K}{\mathbf{c}}, V{\mathbf{c}}),\ (\mathbf{K}{\mathbf{c}}, V{\mathbf{c}}),\ (\mathbf{K}{\mathbf{d}}, V{\mathbf{d}}),\ (\mathbf{K}{\mathbf{e}}, V{\mathbf{e}}),\ (\mathbf{K}{\mathbf{f}}, V{\mathbf{f}})\ \ \text{and}\ \ (\mathbf{K}{\mathbf{g}}, V{\mathbf{g}}) $$ where the keys are, $$ \begin{aligned} &K{\mathbf{a}} = 10101100, K{\mathbf{b}} = 10010010, K{\mathbf{c}} = 10001010, K{\mathbf{d}} = 11100110,\\ &K{\mathbf{e}} = 11110101, K{\mathbf{f}} = 10001011, K{\mathbf{g}} = 00011111. \end{aligned} $$ The leaf levels are as follows; $\text{lvl}(\mathbf{L}{\mathbf{a}}) = 2$, $\text{lvl}(\mathbf{L}{\mathbf{b}}) = 4$, $\text{lvl}(\mathbf{L}{\mathbf{c}}) = 4$, $\text{lvl}(\mathbf{L}{\mathbf{d}}) = 3$, $\text{lvl}(\mathbf{L}{\mathbf{e}}) = 2$, $\text{lvl}(\mathbf{L}{\mathbf{f}}) = 3$ and $\text{lvl}(\mathbf{L}{\mathbf{g}}) = 3$. !Figure 6: An SMT of 7 key-value pairs As illustrated, keys basically determine the shape of the SMT. They dictate where respective leaves must be placed when building the SMT. The main determining factor of the SMT shape is in fact the common key-bits among the keys. For instance, the reason why the leaves $\mathbf{L}{\mathbf{b}}$ and $\mathbf{L}{\mathbf{c}}$ have the largest leaf level 4 is because the two leaves have the longest string of common key-bits "010" in the SMT of Figure 6 above. This explains why different leaves in SMTs can have different levels. The height of a Merkle tree refers to the largest number of edges traversed when navigating from the root to any leaf. Since all leaves are of the same level in Merkle trees, the concept of a height coincide with that of the level of a leaf for Merkle trees. But this is not the case for SMTs. Since leaf levels differ from one leaf to another in SMTs, the height of an SMT is not the same as the leaf level. Rather, the height of an SMT is defined as the largest leaf level among the various leaf levels of leaves on the SMT. For instance, the height of the SMT depicted in the figure above, is 4. Now, since all keys have the same fixed key-length, they not only influence SMT leaf levels and shapes, but also restrict SMT heights to the fixed key-length. The maximum height of an SMT is the maximum key-length imposed on all keys. Remaining keys In a general Sparse Merkle tree (SMT), values are stored at their respective leaf-nodes. But a leaf node $\mathbf{L}{\mathbf{x}}$ not only stores a value, $V\mathbf{{x}}$, but also the key-bits that are left unused in the navigation from the root to $\mathbf{L}{\mathbf{x}}$. These unused key-bits are called the remaining key, and are denoted by $\text{RK}{\mathbf{x}}$ for the leaf node $\mathbf{L}{\mathbf{x}}$. Consider again the SMT of the 7 key-value pairs depicted in the figure above. The remaining keys of each of the 7 leaves in the SMT are as follows; $\text{RK}{\mathbf{a}} = 110101$, $\text{RK}{\mathbf{b}} = 1001$, $\text{RK}{\mathbf{c}} = 0001$, $\text{RK}{\mathbf{d}} = 00111$, $\text{RK}{\mathbf{e}} = 101111$, $\text{RK}{\mathbf{f}} = 10001$ and $\text{RK}{\mathbf{g}} = 11000$. Fake-leaf attack Note that the above simplified design of binary SMTs, based on key-value pairs, presents some problems. The characteristic of binary SMTs having leaves at different levels can be problematic to verifiers, especially when carrying out a simple Merkle proof. Scenario: Fake SMT leaf What if the verifier is presented with a fake leaf? Consider the below figure, showing a binary SMT with a branch $\mathbf{{B}{ab}}$ and its children $\mathbf{L{a}}$ and $\mathbf{L{b}}$ hidden from the verifier's sight. That is, suppose the verifier is provided with the following information; - The key-value $(K{\mathbf{fk}}, V\mathbf{{fk}})$, where $K{\mathbf{fk}} = 11010100$ and $V\mathbf{{fk}} = \mathbf{L{a}} \| \mathbf{L{b}}$. - The root $\mathbf{{root}{ab..f}}$, the number of levels to root, and the siblings $\mathbf{{S}{cd}}$ and $\mathbf{{S}{ef}}$. That is, the Attacker claims that some $V\mathbf{{fk}}$ is stored at $\mathbf{L{fk}} := \mathbf{{B}{ab}}$. Verifier is unaware that $V\mathbf{{fk}}$ is in fact the concatenated value of the hidden real leaves, $\mathbf{L{a}}$ and $\mathbf{L{b}}$, that are children of the supposed leaf $\mathbf{L{fk}}$. i.e., Verifier does not know that leaf $\mathbf{L{fk}}$ is in fact a branch. !Figure 7: MPT - Fake Leaf Attack So then, the verifier being unaware that $\mathbf{L{fk}}$ is not a properly constructed leaf, starts verification as follows; 1. He uses the key $K{\mathbf{fk}}$ to navigate the tree until locating the supposed leaf $\mathbf{L{fk}}$. 2. He computes $\mathbf{H}(V\mathbf{{fk}})$ and sets it as $\tilde{\mathbf{L}}\mathbf{fk} := \mathbf{H}(V\mathbf{{fk}})$. 3. Then takes the sibling $\mathbf{{S}{cd}}$ and calculates $\tilde{ \mathbf{B}}\mathbf{fkcd} = \mathbf{H} \big(\tilde{\mathbf{L}}\mathbf{fk} \| \mathbf{S}{\mathbf{cd}} \big)$. 4. And then, uses $\tilde{ \mathbf{B}}\mathbf{fkcd}$ to compute the root, $\tilde{ \mathbf{root}}\mathbf{ab..f} = \mathbf{H} \big(\tilde{ \mathbf{B}}\mathbf{fkcd}\| \mathbf{S}{\mathbf{ef}} \big)$. The question is: "Does the fake leaf $\mathbf{L{fk}}$ pass the verifier's Merkle proof or not?" Or, equivalently: "Is $\tilde{ \mathbf{root}}\mathbf{ab..f}$ equal to $\mathbf{root}{\mathbf{ab..f}}$?" Since the actual branch $\mathbf{{B}{ab}}$ is by construction the hash, $\mathbf{H}(\mathbf{L{a}} \| \mathbf{L{b}})$, then $\mathbf{{B}{ab}} = \tilde{\mathbf{L}}\mathbf{fk}$. The parent branch ${\mathbf{B}}\mathbf{abcd}$ also, being constructed as the hash, $\mathbf{H} \big(\mathbf{B}\mathbf{ab}\| { \mathbf{S}}\mathbf{cd} \big)$, should be equal to $\mathbf{H} \big(\tilde{\mathbf{L}}\mathbf{fk} \| \mathbf{S}\mathbf{cd} \big) = \tilde{ \mathbf{B}}\mathbf{fkcd}$. As a result, $\mathbf{root}{\mathbf{ab..f}} = \mathbf{H} \big({\mathbf{B}}\mathbf{abcd} \| \mathbf{S}\mathbf{ef} \big) = \mathbf{H} \big(\tilde{ \mathbf{B}}\mathbf{fkcd}\| \mathbf{S}\mathbf{ef} \big) = \tilde{ \mathbf{root}}\mathbf{ab..f}$. Therefore, the fake leaf $\mathbf{L{fk}}$ passes the Merkle proof. Solution to the Fake-leaf attack In order to circumvent the Fake-Leaf Attack we modify how the binary SMTs are built. Here's the trick: When building binary SMTs, differentiate between how leaves are hashed and how branches are hashed. That is, use two different hash functions; one hash function to hash leaves, denote it by $\mathbf{H}{\mathbf{leaf}}$, and the other function for hashing non-leaf nodes, denote it by $\mathbf{H}{\mathbf{noleaf}}$. How does this prevent the Fake-leaf attack? Reconsider now, the Scenario A, given above. Recall that the Attacker provides the following; - The key-value $(K{\mathbf{fk}}, V\mathbf{{fk}})$, where $K{\mathbf{fk}} = 11010100$ and $V\mathbf{{fk}} = \mathbf{L{a}} \| \mathbf{L{b}}$. - The root $\mathbf{{root}{ab..f}}$, the number of levels to root, and the siblings $\mathbf{{S}{cd}}}$ and $\mathbf{{S}{ef}}$. The verifier suspecting no foul, uses $K{\mathbf{fk}} = 11010100$ to navigate the tree until he finds $V\mathbf{{fk}}$ stored at $\mathbf{L{fk}} := \mathbf{{B}{ab}}$. He subsequently starts the Merkle proof by hashing the value $\tilde{V}\mathbf{fk}}$ stored at the located leaf. Since, this computation amounts to forming a leaf, he uses the leaf-hash function, $\mathbf{H}{\mathbf{leaf}}$. - He then sets $\tilde{\mathbf{L}}\mathbf{fk} := \mathbf{H}{\mathbf{leaf}} \big(V\mathbf{{fk}} \big) = \mathbf{H}{\mathbf{leaf}} \big(\mathbf{L{a}} \| \mathbf{L{b}} \big)$. - And further computes $\tilde{ \mathbf{B}}\mathbf{fkcd} = \mathbf{H}{\mathbf{noleaf}} \big(\tilde{\mathbf{L}}\mathbf{fk} \| \mathbf{S}{\mathbf{cd}} \big)$. - Again, calculates the root, $\tilde{ \mathbf{root}}\mathbf{ab..f} = \mathbf{H}{\mathbf{noleaf}} \big(\tilde{ \mathbf{B}}\mathbf{fkcd}\| \mathbf{S}{\mathbf{ef}} \big)$. But the actual branch $\mathbf{{B}{ab}}$ was constructed with the no-leaf-hash function, $\mathbf{H}{\mathbf{noleaf}}$. That is, $$ \mathbf{{B}{ab}} = \mathbf{H}{\mathbf{noleaf}} (\mathbf{L{a}} \| \mathbf{L{b}}) \neq \mathbf{H}{\mathbf{leaf}} \big(\mathbf{L{a}} \| \mathbf{L{b}} \big) = \tilde{\mathbf{L}}\mathbf{fk}. $$ The parent branch ${\mathbf{B}}\mathbf{abcd}}$ also, was constructed as, ${\mathbf{B}}\mathbf{abcd}} = \mathbf{H}{\mathbf{noleaf}} \big(\mathbf{B}\mathbf{ab}}\| {\mathbf{S}}\mathbf{cd}}\big)$. Since the hash functions used are collision-resistant, ${\mathbf{B}}\mathbf{abcd}}$ cannot be equal to $\mathbf{H}{\mathbf{noleaf}} \big(\tilde{\mathbf{L}}\mathbf{fk} \| \mathbf{S}{\mathbf{cd}} \big) = \tilde{ \mathbf{B}}\mathbf{fkcd}$. Consequently, $\mathbf{root}{\mathbf{ab..f}} \neq \tilde{ \mathbf{root}}\mathbf{ab..f}$. Therefore, the Merkle Proof fails. Non-binding key-value pairs Whenever the verifier needs to check inclusion of the given key-value pair $(K{\mathbf{x}}, \text{V}{\mathbf{x}})$ in a binary SMT identified by the $\mathbf{{root}{a..x}}$, he first navigates the SMT in order to locate the leaf $\mathbf{{L}{x}}$ storing $\text{V}{\mathbf{x}}$, and thereafter carries out two computations. Both computations involve climbing the tree from the located leaf $\mathbf{{L}{x}}$ back to the root, $\mathbf{{root}{a..x}}$. And the two computations are; 1. Checking correctness of the key $K{\mathbf{x}}$. That is, verifier takes the Remaining Key, $\text{RK}{\mathbf{x}}$, and reconstructs the key $K{\mathbf{x}}$ by concatenating the key bits used to navigate to $\mathbf{{L}{x}}$ from $\mathbf{{root}{a..x}}$, in the reverse order. Suppose the number of levels to root is 3, and the least-significant bits used for navigation are $\text{kb}\mathbf{2}$, $\text{kb}\mathbf{1}$ and $\text{kb}\mathbf{0}$. In order to check key-correctness, verifier takes the remaining key RK and, - Concatenates $\text{kb}\mathbf{2}$ and gets $\text{ } \text{RK} \| \text{kb}\mathbf{2}$, - Concatenates $\text{kb}\mathbf{1}$ then gets $\text{ } \text{RK} \| \text{kb}\mathbf{2} \| \text{kb}\mathbf{1}$, - Concatenates $\text{kb}\mathbf{0}$ and gets $\text{ } \text{RK} \| \text{kb}\mathbf{2} \| \text{kb}\mathbf{1} \| \text{kb}\mathbf{0}$. He then sets $\tilde{K}{\mathbf{x}} := \text{RK} \| \text{kb}\mathbf{2} \| \text{kb}\mathbf{1} \| \text{kb}\mathbf{0}$, and checks if $\tilde{K}{\mathbf{x}}$ equals $K{\mathbf{x}}$. 2. The Merkle proof: That is, checking whether the value stored at the located leaf $\mathbf{{L}{x}}$ was indeed included in computing the root, $\mathbf{{root}{a..x}}$. This computation was illustrated several times in the above discussions. Note that the key-correctness and the Merkle proof are simultaneously carried out. Example: Indistinguishable leaves Suppose a binary SMT contains a key-value pair $(K{\mathbf{d}}, V\mathbf{{d}})$ at the leaf $\mathbf{L{d}}$, where $K{\mathbf{d}} = 11100110$. That is, $\mathbf{L{d}} := \mathbf{H{leaf}}(V\mathbf{{d}})$. Note that, when building binary SMTs, it is permissible to have another key-value pair $(K{\mathbf{x}}, V\mathbf{{x}})$ in the same tree with $V\mathbf{{x}} = V\mathbf{{d}}$. An Attacker can pick the key-value pair $(K{\mathbf{x}}, V\mathbf{{x}})$ such that $V\mathbf{{x}} = V\mathbf{{d}}$ and $K{\mathbf{x}} = 10100110$. And, with the above design, it means $\mathbf{L{x}} = \mathbf{H{leaf}}(V\mathbf{{x}}) = \mathbf{H{leaf}}(V\mathbf{{d}}) = \mathbf{L{d}}$. Consider the below figure and suppose the Attacker provides the following data; !Non-binding Key-Value Pairs - The key-value $(K{\mathbf{x}}, V\mathbf{{x}})$, where $K{\mathbf{x}} = 10100110$ and $V\mathbf{{x}} = V\mathbf{d}$. - The root, $\mathbf{{root}{a..x}}$, the number of levels to root = 3, and the siblings $\mathbf{{B}{bc}}$, $\mathbf{L{a}}$ and $\mathbf{{S}{efg}}$. The verifier uses the least-significant key bits; $\text{kb}\mathbf{0} = 0$, $\text{kb}\mathbf{1} = 1$ and $\text{kb}\mathbf{2} = 1$; to navigate the tree and locate the leaf $\mathbf{L{x}}$ which is positioned at $\mathbf{L{d}}$. In order to ensure that $\mathbf{L{x}}$ actually stores the value $V\mathbf{{x}}$; The verifier first checks key-correctness. He takes the remaining key $\text{RK} = 10100$ and, 1. Concatenates $\text{kb}\mathbf{2} = 1$, and gets $\text{ } \text{RK} \| \text{kb}\mathbf{2} = 10100 \|1$, 2. Concatenates $\text{kb}\mathbf{1} = 0$ to get $\text{ } \text{RK} \| \text{kb}\mathbf{2} \| \text{kb}\mathbf{1} = 10100 \|1\|0$, 3. Concatenates $\text{kb}\mathbf{0} = 0$, yielding $\text{ } \text{RK} \| \text{kb}\mathbf{2} \| \text{kb}\mathbf{1} \| \text{kb}\mathbf{0} = 10100 \|1\|0\|0 = 10100110$. He sets $\tilde{K}{\mathbf{x}} := 10100 \|1\|0\|0 = 10100110$. Since $\tilde{K}{\mathbf{x}}$ equals $K{\mathbf{x}}$, the verifier concludes that the supplied key is correct. As the verifier climbs the tree to test key-correctness, he concurrently checks if the value $V\mathbf{{x}}$ is included in the SMT identified by the given root, $\mathbf{{root}{a..x}}$. That is, he executes the following computations; - He computes the hash of $V\mathbf{{x}}$ and sets it as, $\tilde{\mathbf{L}}\mathbf{x}:= \mathbf{H{leaf}}(V\mathbf{{x}})$. - Then he uses $\mathbf{{B}{bc}}$ to compute, $\tilde{\mathbf{B}}\mathbf{bcd} = \mathbf{H{noleaf}}(\mathbf{{B}{bc}} \|\tilde{\mathbf{L}}\mathbf{x})$. - He also uses $\mathbf{L{a}}$ to compute, $\tilde{\mathbf{B}}\mathbf{abcd} = \mathbf{H{noleaf}}(\mathbf{L{a}} \| \tilde{\mathbf{B}}\mathbf{bcd})$. - He further calculates, $\tilde{\mathbf{root}}\mathbf{abcd} = \mathbf{H{noleaf}}(\tilde{\mathbf{B}}\mathbf{abcd} \| \mathbf{{S}{efg}})$. Next, the verifier checks if $\tilde{\mathbf{root}}\mathbf{abcd}$ equals $\mathbf{root}\mathbf{abcd}$. Since $V\mathbf{{x}} = V\mathbf{{d}}$, it follows that all the corresponding intermediate values to the root are equal; - $\mathbf{L{d}} = \mathbf{H{leaf}}(V\mathbf{{d}}) = \mathbf{H{leaf}}(V\mathbf{{x}}) = \tilde{\mathbf{L}}\mathbf{x}$, - $\mathbf{B}\mathbf{bcd} = \mathbf{H{noleaf}}(\mathbf{{B}{bc}} \| \mathbf{L}\mathbf{d}) = \mathbf{H{noleaf}}(\mathbf{{B}{bc}} \|\tilde{\mathbf{L}}\mathbf{x}) = \tilde{\mathbf{B}}\mathbf{bcd}$, - $\mathbf{B}\mathbf{abcd} = \mathbf{H{noleaf}}(\mathbf{L{a}} \| \mathbf{B}\mathbf{bcd}) = \mathbf{H{noleaf}}(\mathbf{L{a}} \| \tilde{\mathbf{B}}\mathbf{bcd}) = \tilde{\mathbf{B}}\mathbf{abcd}$, - $\mathbf{root}\mathbf{abcd} = \mathbf{H{noleaf}}(\mathbf{B}\mathbf{abcd} \| \mathbf{{S}{efg}}) = \mathbf{H{noleaf}}(\tilde{\mathbf{B}}\mathbf{abcd} \| \mathbf{{S}{efg}}) = \tilde{\mathbf{root}}\mathbf{abcd}$. The verifier therefore concludes that the key-value pair $(K{\mathbf{x}}, V\mathbf{{x}})$ is in the SMT, when it is not. Why is this attack successful? Note that equality of values, $V\mathbf{{x}} = V\mathbf{{d}}$, associated with two distinct keys, has nothing to do with the efficacy of this attack. In fact, for all practical purposes, it should be permissible for distinct leaves to store any value, irrespective of whether other leaves store an equivalent value or not. The downfall of our binary SMTs design, thus far, is that it does not give the verifier any equation that relates or ties the keys to their associated values. In other words, the attack succeeds simply because the key-value pairs (as 'committed' values) are not binding. Solution to the non-binding key-value problem The solution to this problem is straightforward, and it is to build the binary SMTs in such a way that the key-value pairs are binding. This means, create a relationship between the keys and their associated values, so that the verifier can simply check if this relationship holds true. In order to ensure that checking such a relationship blends with the usual proof machinery, one has two options. The naÃ¯ve solution, which involves the keys, is one option. The NaÃ¯ve solution The naÃ¯ve solution is to simply include keys in the argument of the hash function, when forming leaves. That is, when building a binary SMT, one includes a key-value pair $(K{\mathbf{x}}, V\mathbf{{x}})$ by setting the leaf $\mathbf{L{x}}$ to be the hash of both the value and the key; $$ \mathbf{L{x}} = \mathbf{H{leaf}}(K{\mathbf{x}} \| V\mathbf{{x}}) $$ Does this change remedy the non-binding problem? Suppose $(K{\mathbf{x}}, V\mathbf{{x}})$ and $(K{\mathbf{z}}, V\mathbf{{z}})$ are two key-value pairs such that $V\mathbf{{x}} = V\mathbf{{z}}$, while $K\mathbf{{x}}$ and $K\mathbf{{z}}$ differ only in one of the most-significant bits. Since the hash functions used are collision-resistant, it follows that $$ \mathbf{L{x}} = \mathbf{H{leaf}}(K{\mathbf{x}} \| V\mathbf{{x}}) \neq \mathbf{H{leaf}}(K{\mathbf{z}} \| V\mathbf{{z}}) = \mathbf{L{z}} $$ Consequently, although the key-value pairs $(K{\mathbf{x}}, V\mathbf{{x}})$ and $(K{\mathbf{z}}, V\mathbf{{z}})$ might falsely pass the key-correctness check, they do not pass the Merkle proof test. And this is because, collision-resistance also guarantees that the following series of inequalities hold true; $$ \begin{aligned} \mathbf{L{x}} = \mathbf{H{leaf}}(K{\mathbf{x}} \| V\mathbf{{x}}) \neq \mathbf{H{leaf}}(K{\mathbf{z}} \| V\mathbf{{z}}) = \mathbf{L{z}} \\ \\ \mathbf{B{bx}} = \mathbf{H{noleaf}}(\mathbf{S}\mathbf{{b}} \| \mathbf{L}{\mathbf{x}}) \neq \mathbf{H{noleaf}}(\mathbf{S}'\mathbf{{b}} \| \mathbf{L}{\mathbf{z}}) = \mathbf{B{bz}} \\ \\ \mathbf{B{abx}} = \mathbf{H{noleaf}}(\mathbf{S}\mathbf{{a}} \| \mathbf{B{bx}}) \neq \mathbf{H{noleaf}}(\mathbf{S}'\mathbf{{a}} \| \mathbf{B{bz}}) = \mathbf{B{abz}} \end{aligned} $$ where; $\mathbf{S}\mathbf{{b}}$ is a sibling to $\mathbf{L}{\mathbf{x}}$ and $\mathbf{S}\mathbf{{a}}$ is a sibling to $\mathbf{B{bx}}$, making $\mathbf{B{bx}}$ and $\mathbf{B{abx}}$ branches traversed while climbing the tree from $\mathbf{L{x}}$ to root; Similarly, $\mathbf{S}'\mathbf{{b}}$ is a sibling to $\mathbf{L}{\mathbf{z}}$, while $\mathbf{S}'\mathbf{{a}}$ is a sibling to $\mathbf{B{bx}}$, also making $\mathbf{B{bz}}$ and $\mathbf{B{abz}}$ branches traversed while climbing the tree from $\mathbf{L{z}}$ to root. The only chance for the Merkle proof to pass is if the key-value pairs $(K{\mathbf{x}}, V\mathbf{{x}})$ and $(K{\mathbf{z}}, V\mathbf{{z}})$ are distinct and are individually on the same SMT. The inclusion of keys, in the argument of the hash functions, therefore ensures that leaves $\mathbf{L{x}}$ and $\mathbf{L{z}}$ are distinguishable. And most importantly, that key-value pairs in our SMTs are now binding. A better solution The other solution, which is much more apt than the NaÃ¯ve option, utilises the remaining keys when forming leaves. Since levels to root is related to the Remaining Key (RK) notion, a much more apt solution is to rather include the remaining key, $\text{RK}\mathbf{x}$, as the argument to the hash function, instead of the whole key $K{\mathbf{x}}$. That is, for a key-value pair $(K{\mathbf{x}}, V\mathbf{{x}})$, one sets the leaf $\mathbf{L{x}}$ to be the hash of both the value and the remaining key; $$ \mathbf{L{x}} = \mathbf{H{leaf}}(\text{RK}\mathbf{x} \| \text{V}\mathbf{{x}}). $$ With this strategy, the verifier needs the remaining key $\text{RK}\mathbf{{x}}$, instead of the whole key, in order to carry out a Merkle proof. So he adjusts the Merkle proof by; - Firstly, picking the correct hash function $\mathbf{H{leaf}}$ for leaves, - Secondly, concatenating the value $V{\mathbf{x}}$ stored at the leaf $L{\mathbf{x}}$ and the remaining key $\text{RK}\mathbf{{x}}$, instead of the whole key $K{\mathbf{x}}$, - Thirdly, hashing the concatenation $\mathbf{H{leaf}}(\text{RK}\mathbf{x} \| \text{V}\mathbf{{x}}) =: \mathbf{L{x}}$. This approach not only ensures that key-value pairs in our SMTs are now binding, but also implicitly 'encodes' the levels to root in the leaf. The strategy of using the $\text{RK}\mathbf{x}$ instead of the key $K{\mathbf{x}}$, coupled with hashing leaves and branches differently, yields sound verification. Hiding values It is often necessary to ensure that a commitment scheme has the hiding property. And this can be achieved by storing hashes of values, as opposed to plainly storing the values. A leaf therefore is henceforth constructed in two steps; - Firstly, for a key-value pair $(K{\mathbf{x}}, V\mathbf{{x}})$, compute the hash the value $V\mathbf{{x}}$, $$ \text{Hashed Value} = \text{HV}\mathbf{{x}} = \mathbf{H{noleaf}}(V\mathbf{{x}}) $$ - Secondly, form the leaf containing $V\mathbf{{x}}$, as follows, $$ \mathbf{L{x}} = \mathbf{H{leaf}}(\text{RK}\mathbf{x} \| \text{HV}\mathbf{{x}}) $$ Since it is infeasible to compute the preimage of the hash functions, $\mathbf{H{leaf}}$ and $\mathbf{H{noleaf}}$, computing the hash of the value $V\mathbf{{x}}$ amounts to 'encrypting'. The prover therefore achieves zero-knowledge by providing the pair, $(K{\mathbf{x}}, \text{HV}\mathbf{{x}})$, as the key-value pair instead of the explicit one, $(K{\mathbf{x}}, V\mathbf{{x}})$. The verifier, on the other hand, has to adjust the Merkle proof by starting with; - Firstly, picking the correct hash function $\mathbf{H{leaf}}$ for leaf nodes, - Secondly, concatenating the hashed-value $\text{HV}\mathbf{{x}}$ and the remaining key $\text{RK}\mathbf{{x}}$, - Thirdly, hashing the concatenation in order to form the leaf, $\mathbf{L{x} := \mathbf{H{leaf}}(\text{RK}\mathbf{x} \| \text{HV}\mathbf{{x}})$. Example. (Merkle proof on hidden values) The following example illustrates a Merkle proof when the above strategy is applied. Consider an SMT where the keys are 8-bit long, and the prover commits to the key-value $(K{\mathbf{c}} , \text{HV}\mathbf{c}})$ with $K{\mathbf{c}} = 10010100$. See figure below. !ZK Merkle proof example Since the levels to root is 3, the prover provides; the least-significant key-bits, $\text{kb}0 = 0$, $\text{kb}1 = 0$, $\text{kb}2 = 1$, the stored hashed-value $\text{HV}{\mathbf{c}}$, the root $\mathbf{{root}{a..f}}$, the Remaining Key $\mathbf{ \text{RK}{\mathbf{c}}} = 10010$, and the siblings $\mathbf{{S}{ab}}$, $\mathbf{{L}{d}}$ and $\mathbf{{S}{ef}}$. The verifier first uses the least-significant bits of the key $K{\mathbf{c}} = 10010100$ to navigate the SMT from the root, $\mathbf{{root}{a..f}}$, to the leaf $\mathbf{Lc}$. Then, he executes the following computations; 1. He computes, $\mathbf{Lc} = \mathbf{H{leaf}}\big(\mathbf{ \text{RK}{\mathbf{c}}} \| \text{HV}\mathbf{c} \big) = \mathbf{H{leaf}}(10010 \| \text{HV}\mathbf{c}})$ 2. Then, he uses the sibling $\mathbf{{S}{ab}}$ to compute, $\tilde{ \mathbf{B}}{\mathbf{abc}} := \mathbf{H{noleaf}} \big(\mathbf{{S}{ab}}\|\mathbf{L}{\mathbf{c}} \big)$. 3. Next, he computes, $\tilde{ \mathbf{B}}{\mathbf{abcd}} := \mathbf{H{noleaf}} \big(\tilde{ \mathbf{B}}{\mathbf{abc}} \| \mathbf{L}{\mathbf{d}} \big)$. 4. Now, verifier uses $\tilde{ \mathbf{B}}{\mathbf{abcd}}$ to compute the supposed root, $\tilde{ \mathbf{root}}{\mathbf{ab..f}} := \mathbf{H{noleaf}} \big(\tilde{ \mathbf{B}}{\mathbf{abcd}}\| \mathbf{S}{\mathbf{ef}} \big)$. 5. Checks if $\tilde{ \mathbf{root}}{\mathbf{ab..f}}$ equals ${ \mathbf{root}}{\mathbf{ab..f}}$. The verifier accepts that the key-value pair $(K{\mathbf{c}} , V{\mathbf{c}})$ is in the SMT only if $\tilde{ \mathbf{root}}{\mathbf{ab..f}} = { \mathbf{root}}{\mathbf{ab..f}}$. And he does this without any clue about the exact value $V{\mathbf{c}}$ which is hidden as $\text{HV}{\mathbf{c}}$. # simple-smt.md: Understanding the finer details of how the Storage state machine operates requires a good grasp of the way the zkProver's sparse Merkle trees (SMTs) are constructed. This document explains how these SMTs are built. Consider key-value pair based binary SMTs. The focus here is on explaining how to construct an SMT that represents a given set of key-value pairs. And, for the sake of simplicity, we assume 8-bit key-lengths. A NULL or empty SMT has a zero root. That is, it has no key and no value recorded in it. Similarly, a zero node or NULL node refers to a node that carries no value. A binary SMT with one key-value pair A binary SMT with a single key-value pair $(K{\mathbf{a}}, \text{V}{\mathbf{a}})$, is built as follows. Suppose that $K{\mathbf{a}} = 11010110$. In order to build a binary SMT with this single key-value $(K{\mathbf{a}}, \text{V}{\mathbf{a}})$, 1. One computes the hash $\mathbf{H}(\text{V}{\mathbf{a}})$ of the value $\text{V}{\mathbf{a}}$, 2. Sets the leaf $\mathbf{L}{\mathbf{a}} := \mathbf{H}(\text{V}{\mathbf{a}})$, 3. Sets the sibling leaf as a NULL leaf, simply represented as "$\mathbf{0}$", 4. Computes the root as $\mathbf{root}{a0} = \mathbf{H}(\mathbf{L}{\mathbf{a}} \| \mathbf{0})$, with the leaf $\mathbf{L}{\mathbf{a}}$ on the left because the $\text{lsb}(K{\mathbf{a}}) = 0$. That is, between the two edges leading up to the root, the leaf $\mathbf{L}{\mathbf{a}}$ is on the left edge, while the NULL leaf "$\mathbf{0}$" is on the right. See the below figure for the SMT representing the single key-value pair $(K{\mathbf{a}}, \text{V}{\mathbf{a}})$, where $K{\mathbf{a}} = 11010110$.

!A Single key-value pair SMT{width=400px}

Note that the last nodes in binary SMT branches are generally either leaves or zero-nodes. In the case where the least-significant bit, lsb of $K{\mathbf{a}}$ is 1, the SMT with a single key-value pair $(K{\mathbf{a}}, \text{V}{\mathbf{a}})$ would be a mirror image of what is seen in Figure 3. And its root, $\mathbf{root}{0a} = \mathbf{H}(\mathbf{0}\| \mathbf{L}{\mathbf{a}}) \neq \mathbf{root}{a0}$ because $\mathbf{H}$ is a collision-resistant hash function. This example also explains why we need a zero node. Since all trees used in our design are binary SMTs, a zero node is used as a default sibling for computing the parent node. This helps to differentiate between roots (also between parent nodes) because a root node actually identifies an SMT. Therefore, it is crucial to distinguish between $\mathbf{root}{a0} = \mathbf{H}(\mathbf{L}{\mathbf{a}} \| \mathbf{0})$ and $\mathbf{root}{0a} = \mathbf{H}(\mathbf{0}\| \mathbf{L}{\mathbf{a}})$ because they represent two distinct trees. Binary SMTs with two key-value pairs Consider now SMTs with two key-value pairs, $(K{\mathbf{a}}, \text{V}{\mathbf{a}})$ and $(K{\mathbf{b}}, \text{V}{\mathbf{b}})$. There are three distinct cases of how corresponding SMTs can be built, each determined by the keys, $K{\mathbf{a}}$ and $K{\mathbf{b}}$. Case 1 The keys are such that the $\text{lsb}(K{\mathbf{a}}) = 0$ and the $\text{lsb}(K{\mathbf{b}}) = 1$. Suppose that the keys are given as $K{\mathbf{a}} = 11010110$ and $K{\mathbf{b}} = 11010101$. To build a binary SMT with this two key-values, $(K{\mathbf{a}}, \text{V}{\mathbf{a}})$ and $(K{\mathbf{b}}, \text{V}{\mathbf{b}})$, 1. One computes the hashes, $\mathbf{H}(\text{V}{\mathbf{a}})$ and $\mathbf{H}(\text{V}{\mathbf{b}})$ of the values, $\text{V}{\mathbf{a}}$ and $\text{V}{\mathbf{b}}$, respectively, 2. Sets the leaves, $\mathbf{L}{\mathbf{a}} := \mathbf{H}(\text{V}{\mathbf{a}})$ and $\mathbf{L}{\mathbf{b}} := \mathbf{H}(\text{V}{\mathbf{b}})$, 3. Checks if the $\text{lsb}(K{\mathbf{a}})$ differs from the $\text{lsb}(K{\mathbf{b}})$, 4. Since the $\text{lsb}(K{\mathbf{a}}) = 0$ and the $\text{lsb}(K{\mathbf{b}}) = 1$, it means the two leaves can be siblings, 5. One can then compute the root as, $\mathbf{root}{\mathbf{ab}} = \mathbf{H}(\mathbf{L}{\mathbf{a}} \| \mathbf{L}{\mathbf{b}})$. Note that the leaf $\mathbf{L}{\mathbf{a}}$ is on the left because the $\text{lsb}(K{\mathbf{a}}) = 0$, but $\mathbf{L}{\mathbf{b}}$ is on the right because the $\text{lsb}(K{\mathbf{b}}) = 1$. That is, between the two edges leading up to the $\mathbf{root}{\mathbf{ab}}$, the leaf $\mathbf{L}{\mathbf{a}}$ must be on the edge from the left, while $\mathbf{L}{\mathbf{b}}$ is on the edge from the right. Note that in this particular example, any second key $K{\mathbf{b}}$ with its LSB being 1 would result in the corresponding values being siblings. The algorithm unfolds as we build Merkle trees with more key-value pairs. See the below figure for the SMT representing the two key-value pairs $(K{\mathbf{a}}, \text{V}{\mathbf{a}})$ and $(K{\mathbf{b}}, \text{V}{\mathbf{b}})$, where $K{\mathbf{a}} = 11010110$ and $K{\mathbf{b}} = 11010101$.

!Two key-value pairs SMT - Case 1{width=400px}

Case 2 Both keys end with the same key-bit. That is, the $\text{lsb}(K{\mathbf{a}}) = \text{lsb}(K{\mathbf{b}})$, but their second least-significant bits differ. Suppose that the two keys are given as $K{\mathbf{a}} = 11010100$ and $K{\mathbf{b}} = 11010110$. To build a binary SMT with this two key-values, $(K{\mathbf{a}}, \text{V}{\mathbf{a}})$ and $(K{\mathbf{b}}, \text{V}{\mathbf{b}})$; 1. One computes the hashes, $\mathbf{H}(\text{V}{\mathbf{a}})$ and $\mathbf{H}(\text{V}{\mathbf{b}})$ of the values, $\text{V}{\mathbf{a}}$ and $\text{V}{\mathbf{b}}$, respectively. 2. Sets the leaves, $\mathbf{L}{\mathbf{a}} := \mathbf{H}(\text{V}{\mathbf{a}})$ and $\mathbf{L}{\mathbf{b}} := \mathbf{H}(\text{V}{\mathbf{b}})$. 3. Checks if the $\text{lsb}(K{\mathbf{a}})$ differs from the $\text{lsb}(K{\mathbf{b}})$. Since the $\text{lsb}(K{\mathbf{a}}) = 0$ and the $\text{lsb}(K{\mathbf{b}}) = 0$, it means the two leaves cannot be siblings at this position because it would otherwise mean they share the same tree-address 0, which is not allowed. 4. One continues to check if the second least-significant bits of $K{\mathbf{a}}$ and $K{\mathbf{b}}$ differ. Since the $\text{second lsb}(K{\mathbf{a}}) = 0$ and the $\text{second lsb}(K{\mathbf{b}}) = 1$, it means the two leaves $\mathbf{L}{\mathbf{a}}$ and $\mathbf{L}{\mathbf{b}}$ can be siblings at their respective tree-addresses, 00 and 10. 5. Next is to compute the hash $\mathbf{H}(\mathbf{L}{\mathbf{a}} \| \mathbf{L}{\mathbf{b}})$ and set it as the branch $\mathbf{B}{\mathbf{ab}} := \mathbf{H}(\mathbf{L}{\mathbf{a}} \| \mathbf{L}{\mathbf{b}})$ at the tree-address 0. Note that the leaf $\mathbf{L}{\mathbf{a}}$ is on the left because the $\text{second lsb}(K{\mathbf{a}}) = 0$, while $\mathbf{L}{\mathbf{b}}$ is on the right because the $\text{second lsb}(K{\mathbf{b}}) = 1$. 6. The branch $\mathbf{B}{\mathbf{ab}} := \mathbf{H}(\mathbf{L}{\mathbf{a}} \| \mathbf{L}{\mathbf{b}})$ needs a sibling. Since all the values, $\text{V}{\mathbf{a}}$ and $\text{V}{\mathbf{b}}$, are already represented in the tree at $\mathbf{L}{\mathbf{a}}$ and $\mathbf{L}{\mathbf{b}}$, respectively, one therefore sets a NULL leaf "$\mathbf{0}$" as the sibling leaf to $\mathbf{B}{\mathbf{ab}}$. 7. As a result, it is possible to compute the root as, $\mathbf{root}{\mathbf{ab0}} = \mathbf{H}(\mathbf{B}{\mathbf{ab}} \| \mathbf{0})$. Note that, the branch $\mathbf{B}{\mathbf{ab}}$ is on the left because the $\text{lsb}(K{\mathbf{a}}) = 0$, and $\mathbf{0}$ must therefore be on the right. That is, between the two edges leading up to the $\mathbf{root}{\mathbf{ab0}}$, the branch $\mathbf{B}{\mathbf{ab}}$ must be on the edge from the left, while $\mathbf{0}$ is on the edge from the right. See the below figure depicting the SMT representing the two key-value pairs $(K{\mathbf{a}}, \text{V}{\mathbf{a}})$ and $(K{\mathbf{b}}, \text{V}{\mathbf{b}})$, where $K{\mathbf{a}} = 11010100$ and $K{\mathbf{b}} = 11010110$.

!Two key-value pairs SMT - Case 2{width=450px}

Case 3 The first two least-significant bits of both keys are the same, but their third least-significant bits differ. Suppose that the two keys are given as $K{\mathbf{a}} = 11011000$ and $K{\mathbf{b}} = 10010100$. The process for building a binary SMT with these two key-values, $(K{\mathbf{a}}, \text{V}{\mathbf{a}})$ and $(K{\mathbf{b}}, \text{V}{\mathbf{b}})$ is the same as in Case 2; 1. One computes the hashes, $\mathbf{H}(\text{V}{\mathbf{a}})$ and $\mathbf{H}(\text{V}{\mathbf{b}})$ of the values, $\text{V}{\mathbf{a}}$ and $\text{V}{\mathbf{b}}$, respectively. 2. Sets the leaves, $\mathbf{L}{\mathbf{a}} := \mathbf{H}(\text{V}{\mathbf{a}})$ and $\mathbf{L}{\mathbf{b}} := \mathbf{H}(\text{V}{\mathbf{b}})$. 3. Checks if the $\text{lsb}(K{\mathbf{a}})$ differs from the $\text{lsb}(K{\mathbf{b}})$. Since the $\text{lsb}(K{\mathbf{a}}) = 0$ and the $\text{lsb}(K{\mathbf{b}}) = 0$, it means the two leaves cannot be siblings at this position as it would otherwise mean they share the same tree-address 0, which is not allowed. 4. Next verifier continues to check if the second least-significant bits of $K{\mathbf{a}}$ and $K{\mathbf{b}}$ differ. Since the $\text{second lsb}(K{\mathbf{a}}) = 0$ and the $\text{second lsb}(K{\mathbf{b}}) = 0$, it means the two leaves cannot be siblings at this position, because it would otherwise mean they share the same tree-address 00, which is not allowed. 5. Once again he checks if the third least-significant bits of $K{\mathbf{a}}$ and $K{\mathbf{b}}$ differ. Since the $\text{third lsb}(K{\mathbf{a}}) = 0$ and the $\text{third lsb}(K{\mathbf{b}}) = 1$, it means the two leaves $\mathbf{L}{\mathbf{a}}$ and $\mathbf{L}{\mathbf{b}}$ can be siblings at their respective tree-addresses, 000 and 100. 6. One then computes the hash $\mathbf{H}(\mathbf{L}{\mathbf{a}} \| \mathbf{L}{\mathbf{b}})$, and sets it as the branch $\mathbf{B}{\mathbf{ab}} := \mathbf{H}(\mathbf{L}{\mathbf{a}} \| \mathbf{L}{\mathbf{b}})$ at the tree-address 00. The leaf $\mathbf{L}{\mathbf{a}}$ is on the left because the third $\text{lsb}(K{\mathbf{a}}) = 0$, while $\mathbf{L}{\mathbf{b}}$ is on the right because the third $\text{lsb}(K{\mathbf{b}}) = 1$. 7. The branch $\mathbf{B}{\mathbf{ab}} := \mathbf{H}(\mathbf{L}{\mathbf{a}} \| \mathbf{L}{\mathbf{b}})$ needs a sibling. Since all the values, $\text{V}{\mathbf{a}}$ and $\text{V}{\mathbf{b}}$, are already represented in the tree at $\mathbf{L}{\mathbf{a}}$ and $\mathbf{L}{\mathbf{b}}$, respectively, one therefore sets a NULL leaf "$\mathbf{0}$" as the sibling leaf to $\mathbf{B}{\mathbf{ab}}$. 8. One can now compute the hash $\mathbf{H}(\mathbf{B}{\mathbf{ab}} \| \mathbf{0})$, and set it as the branch $\mathbf{B}{\mathbf{ab0}} := \mathbf{H}(\mathbf{B}{\mathbf{ab}} \| \mathbf{0})$ at the tree-address 0. The hash is computed with the branch $\mathbf{B}{\mathbf{ab}}$ on the left because the second lsb of both keys, $K{\mathbf{a}}$ and $K{\mathbf{b}}$, equals 0. Therefore the NULL leaf "$\mathbf{0}$" must be on the right as an argument to the hash. 9. The branch $\mathbf{B}{\mathbf{ab0}} := \mathbf{H}(\mathbf{B}{\mathbf{ab}} \| \mathbf{0})$ also needs a sibling. For the same reason given above, one sets a NULL leaf "$\mathbf{0}$" as the sibling leaf to $\mathbf{B}{\mathbf{ab0}}$. 10. Now, one is able to compute the root as $\mathbf{root}{\mathbf{ab00}} = \mathbf{H}(\mathbf{B}{\mathbf{ab0}} \| \mathbf{0})$. Note that the hash is computed with the branch $\mathbf{B}{\mathbf{ab0}}$ on the left because the lsb of both keys, $K{\mathbf{a}}$ and $K{\mathbf{b}}$, equals 0. That is, between the two edges leading up to the $\mathbf{root}{\mathbf{ab00}}$, the branch $\mathbf{B}{\mathbf{ab0}}$ must be on the edge from the left, while "$\mathbf{0}$" is on the edge from the right. See the below figure depicting the SMT representing the two key-value pairs $(K{\mathbf{a}}, \text{V}{\mathbf{a}})$ and $(K{\mathbf{b}}, \text{V}{\mathbf{b}})$, where $K{\mathbf{a}} = 11011000$ and $K{\mathbf{b}} = 10010100$.

!Two key-value pairs SMT - Case 3{width=550px}

There are several other SMTs of two key-value pairs $(K{\mathbf{x}}, \text{V}{\mathbf{x}})$ and $(K{\mathbf{z}}, \text{V}{\mathbf{z}})$ that can be constructed depending on how long the strings of the common least-significant bits between $K{\mathbf{x}}$ and $K{\mathbf{z}}$ are. In general, when building an SMT, leaves of key-value pairs with the same least-significant key-bits share the same navigational path only until any of the corresponding key-bits differ. These common strings of key-bits dictate where the leaf storing the corresponding value is located in the tree. # sparse-merkle-tree.md: zkProver's data is stored in the form of a special sparse Merkle tree (SMT), which is a tree that combines the concept of a Merkle tree and that of a Patricia trie. The design is based on how the sparse Merkle trees are constructed and how they store keys and values. !!!tip The content of this document is rather elementary. Experienced developers can fast-forward to later sections and only refer back as the need arises. A typical Merkle tree has leaves, branches and a root. A leaf is a node with no child-nodes, while a branch is a node with child-nodes. The root is therefore the node with no parent-node. See the below figure for an example of how a hash function $\mathbf{H}$ is used to create a Merkle tree recording eight values; $$ \text{V}{\mathbf{a}}, \text{V}{\mathbf{b}}, \text{V}{\mathbf{c}}, \text{V}{\mathbf{d}}, \text{V}{\mathbf{e}}, \text{V}{\mathbf{f}}, \text{V}{\mathbf{g}}, \text{V}{\mathbf{h}} $$!A Merkle tree example Firstly, each leaf is nothing but the hash value $\mathbf{H}(\text{V}{\mathbf{i}})$ of a particular value $\text{V}{\mathbf{i}}$, where $\mathbf{i}$ is an element of the index-set $\{ \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{f}, \mathbf{g}, \mathbf{h} \}$. Secondly, the branch nodes are computed as follows; $$ \begin{aligned} &\mathbf{B}{\mathbf{ab}} = \mathbf{H} \big(\mathbf{H}(\text{V}{\mathbf{a}})\| \mathbf{H}(\text{V}{\mathbf{b}})\big),\ \mathbf{B}{\mathbf{cd}} = \mathbf{H} \big(\mathbf{H}(\text{V}{\mathbf{c}})\| \mathbf{H}(\text{V}{\mathbf{d}})\big), \\ &\mathbf{B}{\mathbf{ef}} = \mathbf{H} \big(\mathbf{H}(\text{V}{\mathbf{e}})\| \mathbf{H}(\text{V}{\mathbf{f}})\big),\ \mathbf{B}{\mathbf{gh}} = \mathbf{H} \big(\mathbf{H}(\text{V}{\mathbf{g}})\| \mathbf{H}(\text{V}{\mathbf{h}})\big), \\ &\mathbf{B}{\mathbf{abcd}} = \mathbf{H} \big(\mathbf{B}{\mathbf{ab}}\| \mathbf{B}{\mathbf{cd}}\big),\ \text{and}\ \mathbf{B}{\mathbf{efgh}} = \mathbf{H} \big(\mathbf{B}{\mathbf{ef}}\| \mathbf{B}{\mathbf{gh}} \big). \end{aligned} $$ Thirdly, the root is computed as $\mathbf{root}{\mathbf{a..h}} = \mathbf{H} \big(\mathbf{B}{\mathbf{abcd}}\| \mathbf{B}{\mathbf{efgh}} \big)$. Leaves that share a parent-node are called siblings. The same terminology applies to branches. For example, $\mathbf{B}{\mathbf{ab}}$ and $\mathbf{B}{\mathbf{cd}}$ are sibling branches because they are branches of the same parent, $\mathbf{B}{\mathbf{abcd}}$. Similarly, $\mathbf{B}{\mathbf{efgh}}$ and $\mathbf{B}{\mathbf{abcd}}$ are sibling branches. Keys and navigating a Merkle tree Note the bits used to label edges in figure above. The strings of these bits are called keys, and they indicated where a node is located in a Merkle tree. Since keys uniquely locate nodes, they are used to navigate from the root to the leaves, and backwards. Suppose one is given the key-value pair $(K{\mathbf{d}} , V{\mathbf{d}})$, where the key is $K{\mathbf{d}} = 10010110$. In order to locate the key-value pair $(K{\mathbf{d}} , V{\mathbf{d}})$ in the Merkle depicted in the figure above, the key $K{\mathbf{d}}$ is read bit-by-bit from the right-most bit to the left-most bit. While traversing the tree from the root downwards, - a zero-key-bit "0" means "follow the edge going to the left", - a key-bit "1" means "follow the edge going to the right". Since $K{\mathbf{d} } = 10010110$, as follows: 1. Read the least-significant bit of $K{\mathbf{d}}$, which is 0, hence traverse the tree to the left, and reach $\mathbf{B{abcd}}$. 2. Then read the second significant key-bit, which is "1" in this case. So take the edge going to the right, reaching $\mathbf{B{cd}}$. 3. Again, read the next key-bit, which is "1", hence follow the edge going to the right, reaching the leaf $\mathbf{H}(V{\mathbf{d}})$. Since $\mathbf{H}(V{\mathbf{d}})$ is a leaf and not a branch, and the navigation was correctly done with respect to the given key $K{\mathbf{d}}$, the $\mathbf{H}(V{\mathbf{d}})$ must be the leaf storing the value $V{\mathbf{d}}$. One can similarly climb the tree, going in the reverse direction, by using the key-bits of the given key in the reverse order. That is, starting with the last key-bit used to reach the leaf and ending with the least-significant bit of the key. The tree-address of the value $V{\mathbf{x}}$, herein refers to the position of the leaf $L{\mathbf{x}} := \mathbf{H}(V{\mathbf{x}})$, denoted by the key-bits used to reach $L{\mathbf{x}}$ but in the reverse order. In the above example, the tree-address of $V{\mathbf{d}}$ is 011. Merkle proof example Merkle trees can be used as commitment schemes. And here is an example that follows the (key,value)-pair approach used in the zkProver. Consider the Merkle tree shown in the figure above. If the prover has committed to a value $\text{V}{\mathbf{f}}$ by appending a new leaf $\mathbf{H}(\text{V}{\mathbf{f}})$ to the Merkle tree as depicted in the above figure, he must then avail the following information, to enable verification of his claim; 1. The Merkle root, denoted by $\mathbf{{root}{a..h}}$, 2. The value $\text{V}{\mathbf{f}}$, and 3. The siblings; $\mathbf{H}(\text{V}{\mathbf{e}})$, $\mathbf{B}{\mathbf{gh}}$ and $\mathbf{B}{\mathbf{abcd}}$. Instead of searching through all hash values stored in the tree, the verifier uses only a few hash values of relevant siblings. That is, three siblings in this case. The verifier then checks the prover's claim by computing the Merkle root as follows; 1. He computes $\mathbf{H}(\text{V}{\mathbf{f}})$, which is the hash of the value $\text{V}{\mathbf{f}}$. 2. Then uses the sibling $\mathbf{H}(\text{V}{\mathbf{e}})$ to compute $\mathbf{H} \big(\mathbf{H}(\text{V}{\mathbf{e}})\|\mathbf{H}(\text{V}{\mathbf{f}}) \big) =: \tilde{ \mathbf{B}}{\mathbf{ef}}$, which should be the same as the branch node $\mathbf{B}{\mathbf{ef}}$. 3. Next, he computes $\mathbf{H} \big(\tilde{ \mathbf{B}}

{\mathbf{ef}}\|\mathbf{B}{\mathbf{gh}} \big) =: \tilde{ \mathbf{B}}{\mathbf{efgh}}$, corresponding to the branch node $\mathbf{B}{\mathbf{efgh}}$. 4. Now, uses $\mathbf{H} \big(\mathbf{B}{\mathbf{abcd}}\| \tilde{ \mathbf{B}}{\mathbf{efgh}} \big) =: \tilde{ \mathbf{root}}{\mathbf{a..h}}$. The Merkle proof is concluded by checking whether $\tilde{ \mathbf{root}}{\mathbf{a\dots h}}$ equals to the publicly known root $\mathbf{root}{\mathbf{a..h}}$. !!!note The symbol tilde denoted $\tilde{\Box}$, is used throughout the document to indicate that the computed value, $\tilde{\Box}$, still needs to be checked, or tested to be true. # check-tx-status.md: Here's how to verify the status of a transaction when using an RPC node as an intermediary step to the Polygon zkEVM network. This guide is for users who send transactions to an RPC node, which in turn relays these transactions to the Polygon zkEVM network. Recommended endpoints After sending a transaction (TX) to the network using the ethsendRawTransaction, use the following endpoints to check the TX status: 1. ethgetTransactionByHash 2. ethgetTransactionReceipt Using ethgetTransactionByHash When checking the TX status using the ethgetTransactionByHash, the result can be either one of the following. (a) If the result is null, it means either the TX doesn't exist in the network or it was discarded. (b) If the result contains some data, then - If the fields blocknum and blockhash are null, it means the TX is still in the pool and is pending. - If the fields blocknum and blockhash are NOT null, it means the TX was mined. Using ethgetTransactionReceipt When checking the TX status using the ethgetTransactionReceipt, again the result can either one of the following. (a) If the result is null, it means either the TX doesn't exist or it is still pending to be mined. In this case, use the ethgetTransactionByHash endpoint to check it. (b) If the result contains some data, it means the TX was already mined. However, - If the field status is 1 (success), it means the TX affected the state as expected. - If the field status is 0 (failure), it means the TX has consumed gas used while processing the TX, but hasn't changed the state as expected. # historical-data.md: Find in here a record of the zkEVMâ€™s historical data. This document records timelines, major milestones, forks, and updates to the zkEVM. Join discussions or ask questions on Telegram: https://t.me/polygonzkevmtechnicalupdates/31 25th April, 2024 Cardona testnet | | Version | | :-------------: | :---: | | Node | v0.6.6 | | Prover | v6.0.0 | | Bridge | v0.4.2 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.6.6 | | Supported ForkIDs | Mainnet: ForkIDs 4, 5, 6, 7 & 8. Cardona: ForkIDs 4, 5, 6, 7, 8 & 9. Testnet (Goerli): ForkIDs 1, 2, 3, 4, 5 & 6 | 9th April, 2024 zkEVM Mainnet | | Version | | :-------------: | :---: | | Node | v0.6.5 | | Prover | v6.0.0 | | Bridge | v0.4.2 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.6.5 | | Supported ForkIDs | Mainnet: ForkIDs 4, 5, 6, 7 & 8. Cardona: ForkIDs 4, 5, 6, 7, 8 & 9. Testnet (Goerli): ForkIDs 1, 2, 3, 4, 5 & 6 | 26th March, 2024 Cardona testnet: Elderberry2 (forkId-9) | | Version | | :-------------: | :---: | | Node | v0.6.4 | | Prover | v6.0.0 | | Bridge | v0.4.2 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.6.4 | | Supported ForkIDs | Mainnet: ForkIDs 4, 5, 6, 7 & 8. Cardona: ForkIDs 4, 5, 6, 7, 8 & 9. Testnet (Goerli): ForkIDs 1, 2, 3, 4, 5 & 6 | 25th March, 2024 zkEVM Mainnet | | Version | | :-------------: | :---: | | Node | v0.6.4 | | Prover | v6.0.0 | | Bridge | v0.4.2 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.6.4 | | Supported ForkIDs | Mainnet: ForkIDs 4, 5, 6, 7 & 8. Cardona: ForkIDs 4, 5, 6, 7, & 8. Testnet (Goerli): ForkIDs 1, 2, 3, 4, 5 & 6 | 24th March, 2024 zkEVM Mainnet | | Version | | :-------------: | :---: | | Node | v0.6.3 | | Prover | v6.0.0 | | Bridge | v0.4.2 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.6.3 | | Supported ForkIDs | Mainnet: ForkIDs 4, 5, 6, 7 & 8. Cardona: ForkIDs 4, 5, 6, 7, & 8. Testnet (Goerli): ForkIDs 1, 2, 3, 4, 5 & 6 | 20th March, 2024 Cardona testnet | | Version | | :-------------: | :---: | | Node | v0.6.3 | | Prover | v5.0.7 | | Bridge | v0.4.2 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.6.3 | | Supported ForkIDs | Mainnet: ForkIDs 4, 5, 6, 7 & 8. Cardona: ForkIDs 4, 5, 6, 7, & 8. Testnet (Goerli): ForkIDs 1, 2, 3, 4, 5 & 6 | 14th March, 2024 zkEVM Elderberry Upgrade - Mainnet | | Version | | :-------------: | :---: | | Node | v0.6.2 | | Prover | v5.0.7 | | Bridge | v0.4.2 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.6.2 | | Supported ForkIDs | Mainnet: ForkIDs 4, 5, 6, 7 & 8. Cardona: ForkIDs 4, 5, 6, 7, & 8. Testnet (Goerli): ForkIDs 1, 2, 3, 4, 5 & 6 | 13th March, 2024 Cardona testnet | | Version | | :-------------: | :---: | | Node | v0.6.2 | | Prover | v5.0.6 | | Bridge | v0.4.2 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.6.2 | | Supported ForkIDs | Mainnet: ForkIDs 4, 5, 6, 7 & 8. Cardona: 4, 5, 6, 7, & 8. Testnet (Goerli): ForkIDs 1, 2, 3, 4, 5 & 6 | 4th March, 2024 Cardona testnet | | Version | | :-------------: | :---: | | Node | v0.6.1 | | Prover | v5.0.3 | | Bridge | v0.4.2 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.6.1 | | Supported ForkIDs | Cardona: ForkIDs 4, 5, 6, 7, & 8. Testnet (Goerli): ForkIDs 1, 2, 3, 4, 5 & 6 | 29th Feb, 2024 Cardona testnet | | Version | | :-------------: | :---: | | Node | v.0.6.1 | | Prover | v5.0.3 | | Bridge | v0.4.2 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.6.1 | | Supported ForkIDs | Cardona: ForkIDs 4, 5, 6, 7, & 8. Testnet (Goerli): ForkIDs 1, 2, 3, 4, 5 & 6 | 29th Feb, 2024 zkEVM Mainnet | | Version | | :-------------: | :---: | | Node | v0.5.13 | | Prover | v4.0.19 | | Bridge | v0.4.2 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.5.13 | | Supported ForkIDs | Mainnet: ForkIDs 4, 5, 6 & 7 | 27th Feb, 2024 zkEVM Mainnet | | Version | | :-------------: | :---: | | Node | v0.5.13 | | Prover | v4.0.17 | | Bridge | v0.4.2 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.5.13 | | Supported ForkIDs | Mainnet: ForkIDs 4, 5, 6 & 7 | 27th Feb, 2024 zkEVM Elderberry Upgrade - Cardona | | Version | | :-------------: | :---: | | Node | v.0.6.0 | | Prover | v5.0.1 | | Bridge | v0.4.1 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.6.0 | | Supported ForkIDs | Cardona: ForkIDs 4, 5, 6, 7, & 8. Testnet (Goerli): ForkIDs 1, 2, 3, 4, 5 & 6 | 21st Feb, 2024 zkEVM Mainnet | | Version | | :-------------: | :---: | | Node | v0.5.13 | | Prover | v4.0.16 | | Bridge | v0.4.0 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.5.13 | | Supported ForkIDs | Mainnet: ForkIDs 4, 5, 6 & 7. Cardona: ForkIDs 4, 5, 6 & 7. Testnet (Goerli): ForkIDs 1, 2, 3, 4, 5 & 6 | 19th Feb, 2024 zkEVM Mainnet | | Version | | :-------------: | :---: | | Node | v0.5.12 | | Prover | v4.0.14 | | Bridge | v0.4.0 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.5.12 | | Supported ForkIDs | Mainnet: ForkIDs 4, 5, 6 & 7. Cardona: 4, 5, 6 & 7. Testnet (Goerli): ForkIDs 1, 2, 3, 4, 5 & 6 | 16th Feb, 2024 zkEVM Mainnet | | Version | | :-------------: | :---: | | Node | v0.5.11 | | Prover | v4.0.13 | | Bridge | v0.4.0 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.5.11 | | Supported ForkIDs | Mainnet: ForkIDs 4, 5, 6 & 7. Cardona: ForkIDs 4, 5, 6 & 7. Testnet (Goerli): ForkIDs 1, 2, 3, 4, 5 & 6 | 13th Feb, 2024 zkEVM mainnet Upgrade: Etrog hardfork This upgrade brings the Polygon zkEVM to being almost a type 2 ZK-EVM. See further details here | | Version | | :-------------: | :---: | | Node | v0.5.8 | | Prover | v4.0.8 | | Bridge | v0.4.0 | | Instruction | https://www.notion.so/polygontechnology/Instructions-zkEVM-Mainnet-Beta-Node-v0-5-8-Prover-v4-0-8-a43b5a50878d4d2f844cdf6d9a6674da | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.5.8 | | Supported ForkIDs | Mainnet & Cardona: ForkIDs 4, 5, 6 & 7. Testnet (Goerli): ForkIDs 1, 2, 3, 4, 5 & 6. | 13th Dec, 2023 zkEVM mainnet update This update included a reduced number of RPC logs and aligned error messages to match geth error messages pertaining to gas estimation for unsigned transactions. | | Version | | :-------------: | :---: | | Node | v0.4.4 | | Prover | v3.0.2 | | Bridge | v0.3.1 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.5.8#::text=Bridge%3A%20v0.4.0-,Changelog,-Version%20v0.5.8%20is | | Supported ForkIDs | Mainnet: ForkIDs 4, 5 & 6. Testnet: ForkIDs 1, 2, 3, 4, 5 & 6. | 23rd Nov, 2023 zkEVM mainnet update The changes made in this update were mainly in the node (Additional CQRS header fix and features in v0.4.0). | | Version | | :-------------: | :---: | | Node | v0.4.1 | | Prover | v3.0.2 | | Bridge | v0.3.0 | | Instruction | https://www.notion.so/polygontechnology/Instructions-zkEVM-Mainnet-Node-v0-4-1-Prover-v3-0-2-7a585f394cd24b90b90283086276533c | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.4.1#::text=Bridge%3A%20v0.3.0-,Changelog,-Version%20v0.4.1%20is | | Supported ForkIDs | Mainnet: ForkIDs 4, 5 & 6. Testnet: ForkIDs 1, 2, 3, 4, 5, 6. | Infrastructure partners were instructed to update, and then use instructions given here. 21st Nov, 2023 zkEVM mainnet update This update brought several significant changes to zkEVM node, bridge, prover infrastructure, including changes to RPC, sequencer, synchroniser, database. It also includes WS improvements, along with other fixes. | | Version | | :-------------: | :---: | | Node | v0.4.0 | | Prover | v3.0.2 | | Bridge | v0.3.0 | | Instruction | https://www.notion.so/polygontechnology/Instructions-zkEVM-Testnet-Node-v0-4-0-Prover-v3-0-2-47751d4c80da4b30a9ef350258deb0ab | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.4.0#::text=Bridge%3A%20v0.3.1-,Changelog,-RPC | | Supported ForkIDs | Mainnet: ForkIDs 4, 5 & 6. Testnet: ForkIDs 1, 2, 3, 4, 5, 6. | 9th Nov, 2023 zkEVM mainnet upgrade: Incaberry hardfork The changes made in this update brought cryptographic optimisations and bug fixes to Polygon zkEVM mainnet. | | Version | | :-------------: | :---: | | Node | v0.3.3 | | Prover | v3.0.0 | | Bridge | v0.2.0 | | Instruction | https://www.notion.so/polygontechnology/Instructions-zkEVM-Mainnet-Node-v0-3-3-Prover-v3-0-0-b14faaf6e2f146f0961e3b1556adec34 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.3.3#::text=Bridge%3A%20v0.2.0-,Changelog,-Version%20v0.3.3%20is | | Supported ForkIDs | Mainnet: ForkIDs 4, 5 & 6. Testnet: ForkIDs 1, 2, 3, 4, 5 & 6. | 7th Nov, 2023 zkEVM mainnet: Upcoming Incaberry hardfork preparation The changes made in this update were the node and prover versions, as well as updates to the RPC, sequencer, and synchroniser. | | Version | | :-------------: | :---: | | Node | v0.3.3 | | Prover | v3.0.0 | | Bridge | v0.2.0 | | Instruction | https://www.notion.so/polygontechnology/Instructions-zkEVM-Mainnet-Node-v0-3-3-Prover-v3-0-0-b14faaf6e2f146f0961e3b1556adec34 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.3.3#::text=Bridge%3A%20v0.2.0-,Changelog,-Version%20v0.3.3%20is | | Supported ForkIDs | Mainnet: ForkIDs 4, 5 & 6. Testnet: ForkIDs 1, 2, 3, 4, 5 & 6. | 24th Oct, 2023 zkEVM testnet upgrade: Incaberry hardfork The changes made in this upgrade were cryptographic optimisations and some bug fixes. | | Version | | :-------------: | :---: | | Node | v0.3.2 | | Prover | v3.0.0 | | Bridge | v0.2.0 | | Instruction | https://www.notion.so/polygontechnology/Instructions-zkEVM-Testnet-Node-v0-3-2-Prover-v3-0-0-18d0b34840874e7cb9ded72e8a9e4e08 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.3.2#::text=Bridge%3A%20v0.2.0-,Changelog,-Version%20v0.3.2%20is | | Supported ForkIDs | Mainnet: ForkIDs 4, 5 & 6. Testnet: ForkIDs 1, 2, 3, 4, 5 & 6. | 26th Sep, 2023 Mainnet beta update The changes made in this update were new version updates of the prover and node. | | Version | | :-------------: | :---: | | Node | v0.3.1 | | Prover | v2.2.2 | | Bridge | v0.2.0 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.3.1#::text=Bridge%3A%20v0.2.0-,Changelog,-Version%20v0.3.1%20is | | Supported ForkIDs | Mainnet: ForkIDs 4 & 5. Testnet: ForkIDs 1, 2, 3, 4, & 5. | 20th Sep, 2023 Mainnet beta Dragonfruit (ForkID5) upgrade The changes made in this update were a range of improvements to the network, including support for the latest Ethereum opcode, PUSH0. | | Version | | :-------------: | :---: | | Node | v0.3.0 | | Prover | v2.2.0 | | Bridge | v0.2.0 | | Instruction | https://www.notion.so/polygontechnology/Mainnet-Beta-UPDATE-zkEVM-Node-version-0-3-0-Sep-6-e150867951cd46c088f6d6c03b2cb5db | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.3.0#::text=Bridge%3A%20v0.2.0-,Changelog,-RPC | 18th Sep, 2023 Mainnet beta event The main changes made in this update were the mainnet beta rehashing and L2 bridge resync. | | Version | | :-------------: | :---: | | Node | v0.3.0 | | Prover | v2.2.0 | | Bridge | v0.2.0 | | Instruction | https://www.notion.so/polygontechnology/zkEVM-mainnet-beta-Instructions-for-permissionless-nodes-to-rehash-the-network-53b525498bc94c03ab02bba621f8d8e3 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.3.0#::text=Bridge%3A%20v0.2.0-,Changelog,-RPC | 6th Sep, 2023 zkEVM mainnet beta update The changes made in this update were new versions of the node and prover. | | Version | | :-------------: | :---: | | Node | v0.3.0 | | Prover | v2.2.0 | | Bridge | v0.2.0 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.3.0#::text=Bridge%3A%20v0.2.0-,Changelog,-RPC | 31st Aug, 2023 zkEVM mainnet update The changes made in this update were new versions of the node and the prover. | | Version | | :-------------: | :---: | | Node | v0.2.5 | | Prover | v2.1.2 | | Bridge | v0.2.0 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.2.5#::text=Bridge%3A%20v0.2.0-,Changelog,-Version%20v0.2.5%20is | 27th Jul, 2023 zkEVM mainnet beta update The changes made in this update were version upgrades for the prover, node and bridge. | | Version | | :-------------: | :---: | | Node | v0.2.0 | | Prover | v2.0.1 | | Bridge | v0.2.0 | | Instruction | https://www.notion.so/polygontechnology/Polygon-zkEVM-Version-Updates-for-Node-Prover-and-Bridge-July-2023-273fe49d869347218923a66fa582aa72 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.2.0#::text=Bridge%3A%20v0.2.0-,Changelog,-%3A | 8th Jun, 2023 Mainnet beta update The changes made in this update are detailed in the change logs. | | Version | | :-------------: | :---: | | Node | v0.1.2 | | Prover | v1.1.4-fork.4 | | Bridge | v0.1.0 | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/v0.1.2#::text=Bridge%3A%20v0.1.0-,Changelog,-%3A | 2nd May, 2023 First node update Corresponding commits found here https://github.com/0xPolygonHermez/zkevm-node/tree/v0.0.7 | | Version | | :-------------: | :---: | | Node | v0.0.7 | | Prover | - | | Bridge | - | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.0.7#::text=1ffaef7-,Changelog,-1ffaef7%20fix%20prover | 25th Apr, 2023 Mainnet update The changes made in this update were new versions of the node and prover containers. The prover had some new config parameters. | | Version | | :-------------: | :---: | | Node | v0.0.6 | | Prover | - | | Bridge | - | | Change logs url | https://github.com/0xPolygonHermez/zkevm-node/releases/tag/v0.0.6#::text=5ed8e4c-,Changelog,-5ed8e4c%20Merge%20pull | Telegram: https://t.me/polygonzkevmtechnicalupdates/31 # index.md: This section of the documentation contains guides on how to connect wallets to the Polygon zkEVM network, deploy new or existing Ethereum smart contracts, and bridge assets between Polygon zkEVM and Ethereum. Developers can herein find guides to setting up an RPC zkNode, spinning up a production zkNode, or even implementing their own fully-fledged zkEVM. Polygon zkEVM is fully compatible with Ethereum to provide a complete EVM-like UX for both Developers and Users. Users do not need any special toolings or wallets in order to build or interact with Polygon zkEVM. Simply switch to the zkEVM RPC and start building on a network with a higher throughput and lower fees. !!!caution Check the list of potential risks associated with the use of Polygon zkEVM in the Disclosures section. # json-rpc.md: --- hide: - toc --- Endpoint details are listed in the playground doc below. Terminal Use the Cardona testnet https://rpc.cardona.zkevm-rpc.com to test the endpoint methods in a terminal. For example: zkevmestimateGasPrice sh curl -X POST \ -H "Content-Type: application/json" \ --data '{"jsonrpc":"2.0","method":"zkevmestimateGasPrice","params":[{"from":"0x0000000000000000000000000000000000000001","to":"0x0000000000000000000000000000000000000002","value":"0x1"}],"id":1}' \ https://rpc.cardona.zkevm-rpc.com Result sh {"jsonrpc":"2.0","id":1,"result":"0xe2ea2f0"} JSON RPC API source ??? "JSON API source" json { "openrpc": "1.0.0-rc1", "info": { "title": "zkEVM Endpoints", "version": "2.0.0" }, "methods": [{ "name": "zkevmconsolidatedBlockNumber", "summary": "Returns the latest block number that is connected to the latest batch verified.", "params": [], "result": { "$ref": "#/components/contentDescriptors/BlockNumber" }, "examples": [{ "name": "example", "description": "", "params": [], "result": { "name": "exampleResult", "description": "", "value": "0x1" } }] }, { "name": "zkevmisBlockVirtualized", "summary": "Returns true if the provided block number is already connected to a batch that was already virtualized, otherwise false.", "params": [{ "name": "blockNumber", "schema": { "$ref": "#/components/contentDescriptors/BlockNumber" } }], "result": { "name": "result", "schema": { "type": "boolean" } }, "examples": [{ "name": "example", "description": "", "params": [], "result": { "name": "exampleResult", "description": "", "value": true } }] }, { "name": "zkevmisBlockConsolidated", "summary": "Returns true if the provided block number is already connected to a batch that was already verified, otherwise false.", "params": [{ "$ref": "#/components/contentDescriptors/BlockNumber" }], "result": { "name": "result", "schema": { "type": "boolean" } }, "examples": [{ "name": "example", "description": "", "params": [], "result": { "name": "exampleResult", "description": "", "value": true } }] }, { "name": "zkevmbatchNumber", "summary": "Returns the latest batch number.", "params": [], "result": { "$ref": "#/components/contentDescriptors/BatchNumber" }, "examples": [{ "name": "example", "description": "", "params": [], "result": { "name": "exampleResult", "description": "", "value": "0x1" } }] }, { "name": "zkevmvirtualBatchNumber", "summary": "Returns the latest virtual batch number.", "params": [], "result": { "$ref": "#/components/contentDescriptors/BatchNumber" }, "examples": [{ "name": "example", "description": "", "params": [], "result": { "name": "exampleResult", "description": "", "value": "0x1" } }] }, { "name": "zkevmverifiedBatchNumber", "summary": "Returns the latest verified batch number.", "params": [], "result": { "$ref": "#/components/contentDescriptors/BatchNumber" }, "examples": [{ "name": "example", "description": "", "params": [], "result": { "name": "exampleResult", "description": "", "value": "0x1" } }] }, { "name": "zkevmbatchNumberByBlockNumber", "summary": "Returns the batch number of the batch connected to the block.", "params": [{ "$ref": "#/components/contentDescriptors/BlockNumber" }], "result": { "$ref": "#/components/contentDescriptors/BatchNumber" }, "examples": [{ "name": "example", "description": "", "params": [], "result": { "name": "exampleResult", "description": "", "value": "0x1" } }] }, { "name": "zkevmgetBatchByNumber", "summary": "Gets a batch for a given number", "params": [{ "$ref": "#/components/contentDescriptors/BatchNumberOrTag" }, { "name": "includeTransactions", "description": "If true it returns the full transaction objects, if false only the hashes of the transactions.", "required": true, "schema": { "title": "isTransactionsIncluded", "type": "boolean" } }], "result": { "$ref": "#/components/contentDescriptors/Batch" }, "examples": [{ "name": "batch without tx details", "description": "Batch without transaction details", "params": [{ "name": "batch number", "value": "0x1" }, { "name": "include txs", "value": "false" }], "result": { "name": "Batch", "value": { "number": "0x1", "coinbase": "0x0000000000000000000000000000000000000001", "stateRoot": "0x0000000000000000000000000000000000000000000000000000000000000001", "globalExitRoot": "0x0000000000000000000000000000000000000000000000000000000000000002", "mainnetExitRoot": "0x0000000000000000000000000000000000000000000000000000000000000003", "rollupExitRoot": "0x0000000000000000000000000000000000000000000000000000000000000004", "localExitRoot": "0x0000000000000000000000000000000000000000000000000000000000000005", "accInputHash": "0x0000000000000000000000000000000000000000000000000000000000000006", "timestamp": "0x642af31f", "sendSequencesTxHash": "0x0000000000000000000000000000000000000000000000000000000000000007", "verifyBatchTxHash": "0x0000000000000000000000000000000000000000000000000000000000000008", "transactions": ["0x0000000000000000000000000000000000000000000000000000000000000009", "0x0000000000000000000000000000000000000000000000000000000000000010", "0x0000000000000000000000000000000000000000000000000000000000000011"] } } }, { "name": "batch with tx detail", "description": "Batch with transaction details", "params": [{ "name": "batch number", "value": "0x1" }, { "name": "include txs", "value": "true" }], "result": { "name": "Batch", "value": { "number": "0x1", "coinbase": "0x0000000000000000000000000000000000000001", "stateRoot": "0x0000000000000000000000000000000000000000000000000000000000000001", "globalExitRoot": "0x0000000000000000000000000000000000000000000000000000000000000002", "mainnetExitRoot": "0x0000000000000000000000000000000000000000000000000000000000000003", "rollupExitRoot": "0x0000000000000000000000000000000000000000000000000000000000000004", "localExitRoot": "0x0000000000000000000000000000000000000000000000000000000000000005", "accInputHash": "0x0000000000000000000000000000000000000000000000000000000000000006", "timestamp": "0x642af31f", "sendSequencesTxHash": "0x0000000000000000000000000000000000000000000000000000000000000007", "verifyBatchTxHash": "0x0000000000000000000000000000000000000000000000000000000000000008", "transactions": [{ "nonce": "0x1", "gasPrice": "0x123456", "gas": "0x59D8", "to": "0x0000000000000000000000000000000000000002", "value": "0x1", "input": "0x", "v": "0xAAA", "r": "0x0000000000000000000000000000000000000000000000000000000000000010", "s": "0x0000000000000000000000000000000000000000000000000000000000000011", "hash": "0x0000000000000000000000000000000000000000000000000000000000000012", "from": "0x0000000000000000000000000000000000000003", "blockHash": "0x0000000000000000000000000000000000000000000000000000000000000013", "blockNumber": "0x1", "transactionIndex": "0x0", "chainId": "0x539", "type": "0x0" }] } } }] }, { "name": "zkevmgetFullBlockByNumber", "summary": "Gets a block with extra information for a given number", "params": [{ "$ref": "#/components/contentDescriptors/BlockNumber" }, { "name": "includeTransactions", "description": "If true it returns the full transaction objects, if false only the hashes of the transactions.", "required": true, "schema": { "title": "isTransactionsIncluded", "type": "boolean" } }], "result": { "name": "getBlockByNumberResult", "schema": { "$ref": "#/components/schemas/FullBlockOrNull" } } }, { "name": "zkevmgetFullBlockByHash", "summary": "Gets a block with extra information for a given hash", "params": [{ "name": "blockHash", "required": true, "schema": { "$ref": "#/components/schemas/BlockHash" } }, { "name": "includeTransactions", "description": "If true it returns the full transaction objects, if false only the hashes of the transactions.", "required": true, "schema": { "title": "isTransactionsIncluded", "type": "boolean" } }], "result": { "name": "getBlockByHashResult", "schema": { "$ref": "#/components/schemas/FullBlockOrNull" } } }, { "name": "zkevmgetNativeBlockHashesInRange", "summary": "Returns the list of native block hashes.", "params": [{ "name": "filter", "schema": { "$ref": "#/components/schemas/NativeBlockHashBlockRangeFilter" } }], "result": { "name": "filter", "schema": { "$ref": "#/components/schemas/NativeBlockHashes" } } }, { "name": "zkevmgetTransactionByL2Hash", "summary": "Returns the information about a transaction requested by transaction l2 hash.", "params": [{ "$ref": "#/components/contentDescriptors/TransactionHash" }], "result": { "$ref": "#/components/contentDescriptors/TransactionResult" } }, { "name": "zkevmgetTransactionReceiptByL2Hash", "summary": "Returns the receipt information of a transaction by its l2 hash.", "params": [{ "$ref": "#/components/contentDescriptors/TransactionHash" }], "result": { "name": "transactionReceiptResult", "description": "returns either a receipt or null", "schema": { "title": "transactionReceiptOrNull", "oneOf": [{ "$ref": "#/components/schemas/Receipt" }, { "$ref": "#/components/schemas/Null" }] } } }, { "name": "zkevmgetExitRootsByGER", "summary": "Gets the exit roots accordingly to the provided Global Exit Root", "params": [{ "$ref": "#/components/schemas/Keccak" }], "result": { "$ref": "#/components/schemas/ExitRoots" }, "examples": [{ "name": "exit roots", "params": [{ "name": "global exit root", "value": "0x0000000000000000000000000000000000000000000000000000000000000001" }], "result": { "name": "Exit Roots", "value": { "blockNumber": "0x1", "timestamp": "0x642af31f", "mainnetExitRoot": "0x0000000000000000000000000000000000000000000000000000000000000002", "rollupExitRoot": "0x0000000000000000000000000000000000000000000000000000000000000003" } } }] }, { "name": "zkevmgetLatestGlobalExitRoot", "summary": "Returns the latest global exit root used in a batch.", "params": [ ], "result": { "name": "GER", "schema": { "$ref": "#/components/schemas/Keccak" } } }, { "name": "zkevmestimateCounters", "summary": "Estimates the transaction ZK Counters", "params": [{ "$ref": "#/components/contentDescriptors/Transaction" }], "result": { "name": "counters", "description": "The counters used, limits and revert info when tx reverted", "schema": { "$ref": "#/components/schemas/ZKCountersResponse" } } }, { "name": "zkevmestimateFee", "summary": "Estimates the transaction Fee following the effective gas price rules", "params": [{ "$ref": "#/components/contentDescriptors/Transaction" }], "result": { "name": "fee", "description": "The amount of the fee", "schema": { "$ref": "#/components/schemas/Integer" } } }, { "name": "zkevmestimateGasPrice", "summary": "Estimates the transaction Gas Price following the effective gas price rules", "params": [{ "$ref": "#/components/contentDescriptors/Transaction" }], "result": { "name": "gasPrice", "description": "The amount of gas price", "schema": { "$ref": "#/components/schemas/Integer" } } }], "components": { "contentDescriptors": { "BlockNumber": { "name": "blockNumber", "required": true, "schema": { "$ref": "#/components/schemas/BlockNumber" } }, "BatchNumber": { "name": "batchNumber", "required": true, "schema": { "$ref": "#/components/schemas/BatchNumber" } }, "BatchNumberOrTag": { "name": "batchNumberOrTag", "required": true, "schema": { "title": "batchNumberOrTag", "oneOf": [{ "$ref": "#/components/schemas/BatchNumber" }, { "$ref": "#/components/schemas/BatchNumberTag" }] } }, "Batch": { "name": "batch", "description": "batch", "required": true, "schema": { "$ref": "#/components/schemas/Batch" } }, "Block": { "name": "block", "summary": "A block", "description": "A block object", "schema": { "$ref": "#/components/schemas/Block" } }, "Transaction": { "required": true, "name": "transaction", "schema": { "$ref": "#/components/schemas/Transaction" } }, "TransactionHash": { "name": "transactionHash", "required": true, "schema": { "$ref": "#/components/schemas/TransactionHash" } }, "TransactionResult": { "name": "transactionResult", "description": "Returns a transaction or null", "schema": { "title": "TransactionOrNull", "oneOf": [{ "$ref": "#/components/schemas/Transaction" }, { "$ref": "#/components/schemas/Null" }] } } }, "schemas": { "Null": { "title": "null", "type": "null", "description": "Null" }, "BatchNumberTag": { "title": "batchNumberTag", "type": "string", "description": "The optional batch height description", "enum": ["earliest", "latest"] }, "Integer": { "title": "integer", "type": "string", "pattern": "^0x[a-fA-F0-9]+$", "description": "Hex representation of the integer" }, "Keccak": { "title": "keccak", "type": "string", "description": "Hex representation of a Keccak 256 hash", "pattern": "^0x[a-fA-F\\d]{64}$" }, "Address": { "title": "address", "type": "string", "pattern": "^0x[a-fA-F\\d]{40}$" }, "BlockHash": { "title": "blockHash", "type": "string", "pattern": "^0x[a-fA-F\\d]{64}$", "description": "The hex representation of the Keccak 256 of the RLP encoded block" }, "BlockNumber": { "title": "blockNumber", "type": "string", "description": "The hex representation of the block's height", "$ref": "#/components/schemas/Integer" }, "FullBlockOrNull": { "title": "fullBlockOrNull", "oneOf": [{ "$ref": "#/components/schemas/FullBlock" }, { "$ref": "#/components/schemas/Null" }] }, "BatchNumber": { "title": "batchNumber", "type": "string", "description": "The hex representation of the batch's height", "$ref": "#/components/schemas/Integer" }, "TransactionHash": { "title": "transactionHash", "type": "string", "description": "Keccak 256 Hash of the RLP encoding of a transaction", "$ref": "#/components/schemas/Keccak" }, "NonceOrNull": { "title": "nonceOrNull", "description": "Randomly selected number to satisfy the proof-of-work or null when its the pending block", "oneOf": [{ "$ref": "#/components/schemas/Nonce" }, { "$ref": "#/components/schemas/Null" }] }, "Nonce": { "title": "nonce", "description": "A number only to be used once", "$ref": "#/components/schemas/Integer" }, "From": { "title": "From", "description": "The sender of the transaction", "$ref": "#/components/schemas/Address" }, "BlockNumberOrNull": { "title": "blockNumberOrNull", "description": "The block number or null when its the pending block", "oneOf": [{ "$ref": "#/components/schemas/BlockNumber" }, { "$ref": "#/components/schemas/Null" }] }, "IntegerOrNull": { "title": "integerOrNull", "oneOf": [{ "$ref": "#/components/schemas/Integer" }, { "$ref": "#/components/schemas/Null" }] }, "AddressOrNull": { "title": "addressOrNull", "oneOf": [{ "$ref": "#/components/schemas/Address" }, { "$ref": "#/components/schemas/Null" }] }, "KeccakOrPending": { "title": "keccakOrPending", "oneOf": [{ "$ref": "#/components/schemas/Keccak" }, { "$ref": "#/components/schemas/Null" }] }, "To": { "title": "To", "description": "Destination address of the transaction. Null if it was a contract create.", "oneOf": [{ "$ref": "#/components/schemas/Address" }, { "$ref": "#/components/schemas/Null" }] }, "BlockHashOrNull": { "title": "blockHashOrNull", "description": "The block hash or null when its the pending block", "$ref": "#/components/schemas/KeccakOrPending" }, "TransactionIndex": { "title": "transactionIndex", "description": "The index of the transaction. null when its pending", "$ref": "#/components/schemas/IntegerOrNull" }, "Batch": { "title": "Batch", "type": "object", "readOnly": true, "properties": { "number": { "$ref": "#/components/schemas/BlockNumber" }, "globalExitRoot": { "$ref": "#/components/schemas/Keccak" }, "mainnetExitRoot": { "$ref": "#/components/schemas/Keccak" }, "rollupExitRoot": { "$ref": "#/components/schemas/Keccak" }, "accInputHash": { "$ref": "#/components/schemas/Keccak" }, "timestamp": { "$ref": "#/components/schemas/Integer" }, "sendSequencesTxHash": { "$ref": "#/components/schemas/TransactionHash" }, "verifyBatchTxHash": { "$ref": "#/components/schemas/TransactionHash" }, "closed": { "title": "closed", "type": "boolean", "description": "True if the batch is already closed, otherwise false" }, "blocks": { "title": "blocksOrHashes", "description": "Array of block objects, or 32 Bytes block hashes depending on the last given parameter", "type": "array", "items": { "title": "blockOrBlockHash", "oneOf": [{ "$ref": "#/components/schemas/Block" }, { "$ref": "#/components/schemas/BlockHash" }] } }, "transactions": { "title": "transactionsOrHashes", "description": "Array of transaction objects, or 32 Bytes transaction hashes depending on the last given parameter", "type": "array", "items": { "title": "transactionOrTransactionHash", "oneOf": [{ "$ref": "#/components/schemas/Transaction" }, { "$ref": "#/components/schemas/TransactionHash" }] } }, "stateRoot": { "$ref": "#/components/schemas/Keccak" }, "coinbase": { "$ref": "#/components/schemas/Address" } } }, "Block": { "title": "Block", "type": "object", "properties": { "number": { "$ref": "#/components/schemas/BlockNumberOrNull" }, "hash": { "$ref": "#/components/schemas/BlockHashOrNull" }, "parentHash": { "$ref": "#/components/schemas/BlockHash" }, "nonce": { "$ref": "#/components/schemas/NonceOrNull" }, "sha3Uncles": { "title": "blockShaUncles", "description": "Keccak hash of the uncles data in the block", "$ref": "#/components/schemas/Keccak" }, "logsBloom": { "title": "blockLogsBloom", "type": "string", "description": "The bloom filter for the logs of the block or null when its the pending block", "pattern": "^0x[a-fA-F\\d]+$" }, "transactionsRoot": { "title": "blockTransactionsRoot", "description": "The root of the transactions trie of the block.", "$ref": "#/components/schemas/Keccak" }, "stateRoot": { "title": "blockStateRoot", "description": "The root of the final state trie of the block", "$ref": "#/components/schemas/Keccak" }, "receiptsRoot": { "title": "blockReceiptsRoot", "description": "The root of the receipts trie of the block", "$ref": "#/components/schemas/Keccak" }, "miner": { "$ref": "#/components/schemas/AddressOrNull" }, "difficulty": { "title": "blockDifficulty", "type": "string", "description": "Integer of the difficulty for this block" }, "totalDifficulty": { "title": "blockTotalDifficulty", "description": "Integer of the total difficulty of the chain until this block", "$ref": "#/components/schemas/IntegerOrNull" }, "extraData": { "title": "blockExtraData", "type": "string", "description": "The 'extra data' field of this block" }, "size": { "title": "blockSize", "type": "string", "description": "Integer the size of this block in bytes" }, "gasLimit": { "title": "blockGasLimit", "type": "string", "description": "The maximum gas allowed in this block" }, "gasUsed": { "title": "blockGasUsed", "type": "string", "description": "The total used gas by all transactions in this block" }, "timestamp": { "title": "blockTimeStamp", "type": "string", "description": "The unix timestamp for when the block was collated" }, "transactions": { "title": "transactionsOrHashes", "description": "Array of transaction objects, or 32 Bytes transaction hashes depending on the last given parameter", "type": "array", "items": { "title": "transactionOrTransactionHash", "oneOf": [{ "$ref": "#/components/schemas/Transaction" }, { "$ref": "#/components/schemas/TransactionHash" }] } }, "uncles": { "title": "uncleHashes", "description": "Array of uncle hashes", "type": "array", "items": { "title": "uncleHash", "description": "Block hash of the RLP encoding of an uncle block", "$ref": "#/components/schemas/Keccak" } } } }, "FullBlock": { "title": "fullBlock", "type": "object", "properties": { "number": { "$ref": "#/components/schemas/BlockNumberOrNull" }, "hash": { "$ref": "#/components/schemas/BlockHashOrNull" }, "parentHash": { "$ref": "#/components/schemas/BlockHash" }, "nonce": { "$ref": "#/components/schemas/NonceOrNull" }, "sha3Uncles": { "title": "blockShaUncles", "description": "Keccak hash of the uncles data in the block", "$ref": "#/components/schemas/Keccak" }, "logsBloom": { "title": "blockLogsBloom", "type": "string", "description": "The bloom filter for the logs of the block or null when its the pending block", "pattern": "^0x[a-fA-F\\d]+$" }, "transactionsRoot": { "title": "blockTransactionsRoot", "description": "The root of the transactions trie of the block.", "$ref": "#/components/schemas/Keccak" }, "stateRoot": { "title": "blockStateRoot", "description": "The root of the final state trie of the block", "$ref": "#/components/schemas/Keccak" }, "receiptsRoot": { "title": "blockReceiptsRoot", "description": "The root of the receipts trie of the block", "$ref": "#/components/schemas/Keccak" }, "miner": { "$ref": "#/components/schemas/AddressOrNull" }, "difficulty": { "title": "blockDifficulty", "type": "string", "description": "Integer of the difficulty for this block" }, "totalDifficulty": { "title": "blockTotalDifficulty", "description": "Integer of the total difficulty of the chain until this block", "$ref": "#/components/schemas/IntegerOrNull" }, "extraData": { "title": "blockExtraData", "type": "string", "description": "The 'extra data' field of this block" }, "size": { "title": "blockSize", "type": "string", "description": "Integer the size of this block in bytes" }, "gasLimit": { "title": "blockGasLimit", "type": "string", "description": "The maximum gas allowed in this block" }, "gasUsed": { "title": "blockGasUsed", "type": "string", "description": "The total used gas by all transactions in this block" }, "timestamp": { "title": "blockTimeStamp", "type": "string", "description": "The unix timestamp for when the block was collated" }, "transactions": { "title": "transactionsOrHashes", "description": "Array of transaction objects, or 32 Bytes transaction hashes depending on the last given parameter", "type": "array", "items": { "title": "transactionOrTransactionHash", "oneOf": [{ "$ref": "#/components/schemas/FullTransaction" }, { "$ref": "#/components/schemas/TransactionHash" }] } }, "uncles": { "title": "uncleHashes", "description": "Array of uncle hashes", "type": "array", "items": { "title": "uncleHash", "description": "Block hash of the RLP encoding of an uncle block", "$ref": "#/components/schemas/Keccak" } } } }, "Transaction": { "title": "transaction", "type": "object", "required": ["gas", "gasPrice", "nonce"], "properties": { "blockHash": { "$ref": "#/components/schemas/BlockHashOrNull" }, "blockNumber": { "$ref": "#/components/schemas/BlockNumberOrNull" }, "from": { "$ref": "#/components/schemas/From" }, "gas": { "title": "transactionGas", "type": "string", "description": "The gas limit provided by the sender in Wei" }, "gasPrice": { "title": "transactionGasPrice", "type": "string", "description": "The gas price willing to be paid by the sender in Wei" }, "hash": { "$ref": "#/components/schemas/TransactionHash" }, "l2Hash": { "$ref": "#/components/schemas/TransactionHash" }, "input": { "title": "transactionInput", "type": "string", "description": "The data field sent with the transaction" }, "nonce": { "title": "transactionNonce", "description": "The total number of prior transactions made by the sender", "$ref": "#/components/schemas/Nonce" }, "to": { "$ref": "#/components/schemas/To" }, "transactionIndex": { "$ref": "#/components/schemas/TransactionIndex" }, "value": { "title": "transactionValue", "description": "Value of Ether being transferred in Wei", "$ref": "#/components/schemas/Keccak" }, "v": { "title": "transactionSigV", "type": "string", "description": "ECDSA recovery id" }, "r": { "title": "transactionSigR", "type": "string", "description": "ECDSA signature r" }, "s": { "title": "transactionSigS", "type": "string", "description": "ECDSA signature s" } } }, "FullTransaction": { "title": "fullTransaction", "type": "object", "required": ["gas", "gasPrice", "nonce"], "properties": { "blockHash": { "$ref": "#/components/schemas/BlockHashOrNull" }, "blockNumber": { "$ref": "#/components/schemas/BlockNumberOrNull" }, "from": { "$ref": "#/components/schemas/From" }, "gas": { "title": "transactionGas", "type": "string", "description": "The gas limit provided by the sender in Wei" }, "gasPrice": { "title": "transactionGasPrice", "type": "string", "description": "The gas price willing to be paid by the sender in Wei" }, "hash": { "$ref": "#/components/schemas/TransactionHash" }, "input": { "title": "transactionInput", "type": "string", "description": "The data field sent with the transaction" }, "nonce": { "title": "transactionNonce", "description": "The total number of prior transactions made by the sender", "$ref": "#/components/schemas/Nonce" }, "to": { "$ref": "#/components/schemas/To" }, "transactionIndex": { "$ref": "#/components/schemas/TransactionIndex" }, "value": { "title": "transactionValue", "description": "Value of Ether being transferred in Wei", "$ref": "#/components/schemas/Keccak" }, "v": { "title": "transactionSigV", "type": "string", "description": "ECDSA recovery id" }, "r": { "title": "transactionSigR", "type": "string", "description": "ECDSA signature r" }, "s": { "title": "transactionSigS", "type": "string", "description": "ECDSA signature s" }, "receipt": { "$ref": "#/components/schemas/Receipt" } } }, "Transactions": { "title": "transactions", "description": "An array of transactions", "type": "array", "items": { "$ref": "#/components/schemas/Transaction" } }, "Receipt": { "title": "receipt", "type": "object", "description": "The receipt of a transaction", "required": ["blockHash", "blockNumber", "contractAddress", "cumulativeGasUsed", "from", "gasUsed", "logs", "logsBloom", "to", "transactionHash", "transactionIndex"], "properties": { "blockHash": { "$ref": "#/components/schemas/BlockHash" }, "blockNumber": { "$ref": "#/components/schemas/BlockNumber" }, "contractAddress": { "title": "ReceiptContractAddress", "description": "The contract address created, if the transaction was a contract creation, otherwise null", "$ref": "#/components/schemas/AddressOrNull" }, "cumulativeGasUsed": { "title": "ReceiptCumulativeGasUsed", "description": "The gas units used by the transaction", "$ref": "#/components/schemas/Integer" }, "from": { "$ref": "#/components/schemas/From" }, "gasUsed": { "title": "ReceiptGasUsed", "description": "The total gas used by the transaction", "$ref": "#/components/schemas/Integer" }, "logs": { "title": "logs", "type": "array", "description": "An array of all the logs triggered during the transaction", "items": { "$ref": "#/components/schemas/Log" } }, "logsBloom": { "$ref": "#/components/schemas/BloomFilter" }, "to": { "$ref": "#/components/schemas/To" }, "transactionHash": { "$ref": "#/components/schemas/TransactionHash" }, "transactionL2Hash": { "$ref": "#/components/schemas/TransactionHash" }, "transactionIndex": { "$ref": "#/components/schemas/TransactionIndex" }, "postTransactionState": { "title": "ReceiptPostTransactionState", "description": "The intermediate stateRoot directly after transaction execution.", "$ref": "#/components/schemas/Keccak" }, "status": { "title": "ReceiptStatus", "description": "Whether or not the transaction threw an error.", "type": "boolean" } } }, "BloomFilter": { "title": "bloomFilter", "type": "string", "description": "A 2048 bit bloom filter from the logs of the transaction. Each log sets 3 bits though taking the low-order 11 bits of each of the first three pairs of bytes in a Keccak 256 hash of the log's byte series" }, "Log": { "title": "log", "type": "object", "description": "An indexed event generated during a transaction", "properties": { "address": { "title": "LogAddress", "description": "Sender of the transaction", "$ref": "#/components/schemas/Address" }, "blockHash": { "$ref": "#/components/schemas/BlockHash" }, "blockNumber": { "$ref": "#/components/schemas/BlockNumber" }, "data": { "title": "LogData", "description": "The data/input string sent along with the transaction", "$ref": "#/components/schemas/Bytes" }, "logIndex": { "title": "LogIndex", "description": "The index of the event within its transaction, null when its pending", "$ref": "#/components/schemas/Integer" }, "removed": { "title": "logIsRemoved", "description": "Whether or not the log was orphaned off the main chain", "type": "boolean" }, "topics": { "$ref": "#/components/schemas/Topics" }, "transactionHash": { "$ref": "#/components/schemas/TransactionHash" }, "transactionIndex": { "$ref": "#/components/schemas/TransactionIndex" } } }, "Topics": { "title": "LogTopics", "description": "Topics are order-dependent. Each topic can also be an array of DATA with 'or' options.", "type": "array", "items": { "$ref": "#/components/schemas/Topic" } }, "Topic": { "title": "topic", "description": "32 Bytes DATA of indexed log arguments. (In solidity: The first topic is the hash of the signature of the event (e.g. Deposit(address,bytes32,uint256))", "$ref": "#/components/schemas/DataWord" }, "DataWord": { "title": "dataWord", "type": "string", "description": "Hex representation of a 256 bit unit of data", "pattern": "^0x([a-fA-F\\d]{64})?$" }, "Bytes": { "title": "bytes", "type": "string", "description": "Hex representation of a variable length byte array", "pattern": "^0x([a-fA-F0-9]?)+$" }, "NativeBlockHashes": { "title": "native block hashes", "description": "An array of hashes", "type": "array", "items": { "$ref": "#/components/schemas/Keccak" } }, "NativeBlockHashBlockRangeFilter": { "title": "NativeBlockHashBlockRangeFilter", "type": "object", "properties": { "fromBlock": { "$ref": "#/components/schemas/BlockNumber" }, "toBlock": { "$ref": "#/components/schemas/BlockNumber" } } }, "ExitRoots": { "title": "ExitRoots", "type": "object", "readOnly": true, "properties": { "blockNumber": { "$ref": "#/components/schemas/BlockNumber" }, "timestamp": { "title": "timestamp", "type": "string", "description": "The unix timestamp of the block mentioned in the blockNumber field" }, "mainnetExitRoot": { "$ref": "#/components/schemas/Keccak" }, "rollupExitRoot": { "$ref": "#/components/schemas/Keccak" } } }, "ZKCountersResponse": { "title": "ZKCountersResponse", "type": "object", "readOnly": true, "properties": { "countersUsed": { "$ref": "#/components/schemas/ZKCountersUsed" }, "countersLimits": { "$ref": "#/components/schemas/ZKCountersLimits" }, "revertInfo": { "$ref": "#/components/schemas/RevertInfo" }, "oocError": { "type": "string" } } }, "ZKCountersUsed": { "title": "ZKCountersUsed", "type": "object", "readOnly": true, "properties": { "gasUsed": { "$ref": "#/components/schemas/Integer" }, "usedKeccakHashes": { "$ref": "#/components/schemas/Integer" }, "usedPoseidonHashes": { "$ref": "#/components/schemas/Integer" }, "usedPoseidonPaddings": { "$ref": "#/components/schemas/Integer" }, "usedMemAligns": { "$ref": "#/components/schemas/Integer" }, "usedArithmetics": { "$ref": "#/components/schemas/Integer" }, "usedBinaries": { "$ref": "#/components/schemas/Integer" }, "usedSteps": { "$ref": "#/components/schemas/Integer" }, "usedSHA256Hashes": { "$ref": "#/components/schemas/Integer" } } }, "ZKCountersLimits":{ "title": "ZKCountersLimits", "type": "object", "readOnly": true, "properties": { "maxGasUsed": { "$ref": "#/components/schemas/Integer" }, "maxUsedKeccakHashes": { "$ref": "#/components/schemas/Integer" }, "maxUsedPoseidonHashes": { "$ref": "#/components/schemas/Integer" }, "maxUsedPoseidonPaddings": { "$ref": "#/components/schemas/Integer" }, "maxUsedMemAligns": { "$ref": "#/components/schemas/Integer" }, "maxUsedArithmetics": { "$ref": "#/components/schemas/Integer" }, "maxUsedBinaries": { "$ref": "#/components/schemas/Integer" }, "maxUsedSteps": { "$ref": "#/components/schemas/Integer" }, "maxUsedSHA256Hashes": { "$ref": "#/components/schemas/Integer" } } }, "RevertInfo":{ "title": "RevertInfo", "type": "object", "readOnly": true, "properties": { "message": { "type": "string" }, "data": { "$ref": "#/components/schemas/Integer" } } } } } } Playground The available methods are detailed in the playground description below. Each method description provides: - Method name and explanation. - Parameters required if any and their details. - Expected return. - Examples. Instructions for use

!!! tip "More information" For more information on the playground, checkout the open RPC playground repo. # quick-start.md: !!!caution Check the list of potential risks associated with the use of Polygon zkEVM in the Disclosures section. Connect to zkEVM mainnet or testnet Add a full network configuration to your wallet in one click.

!img

Chainlist is a site that provides a button to connect your wallet to the Polygon zkEVM mainnet and the Polygon Cardona zkEVM testnet. - Go to the Polygon zkEVM mainnet page and click Connect Wallet to add the Polygon zkEVM mainnet network to your wallet settings. - Go to the Polygon zkEVM Cardona testnet page and click Connect Wallet to add the Polygon zkEVM Cardona testnet configurations to your wallet settings. Manually add network to wallet Connect your wallet to the Polygon zkEVM network of your choice by navigating to the add network settings and entering the network details as shown in the table: | Network | RPC URL | ChainID | Block explorer URL | Currency | | ------- | ------------------------------- | ---------------- | ---------------- | ----- | | Polygon zkEVM | https://zkevm-rpc.com | 1101 | https://zkevm.polygonscan.com/ | ETH | | Cardona zkEVM testnet | https://rpc.cardona.zkevm-rpc.com | 2442 | https://cardona-zkevm.polygonscan.com/ | ETH | Polygon testnet faucet If you plan on developing a dApp for zkEVM, you may want to deploy your test dapp on the Cardona zkEVM testnet. These test tokens enable you to work with Polygon zkEVM without having to spend actual MATIC tokens on the mainnet zkEVM chain. Visit the Polygon MATIC faucet page to learn how to get testnet tokens. Bridging assets to zkEVM Once the wallet is connected, the next step is to bridge crypto assets from Ethereum to zkEVM. Use the Polygon Portal to deposit and swap tokens between the Polygon zkEVM chain (or the Cardona zkEVM testnet) and other chains. Visit the Polygon Portal documentation page for guides on how to connect to and operate the Polygon Portal website. Video demo Here is a video tutorial on how to add Polygon zkEVM testnet to MetaMask and deploy smart contracts:

Your browser does not support the video element.

 # risk-disclosures.md: Polygon zkEVM mainnet beta is provided on an AS-IS and AS-AVAILABLE basis. The documentation contains statements about technical specifications, some of which may relate to future versions of Polygon zkEVM rather than its current implementation. Attack vectors and security - This is a mainnet beta and not a mainnet release of Polygon zkEVM, security audits and assessments are ongoing. Your data and crypto-assets may be at risk as a result of bugs or otherwise. - Polygon zkEVM technology is novel. As such, there may be unanticipated issues and risks associated with your use of the technology. For example, there may be errors that result in losing data or crypto-assets. - Cross-blockchain bridging may be subject to cyberattacks and exploits including, without limitation, hacks that exploit a vulnerability in the software, hardware, systems or equipment associated with any bridge component, smart contracts, and related systems. Network availability and performance As this is a mainnet beta version, Polygon zkEVM may be slow or unavailable from time to time without notice, which could result in unexpected loss of use or data or crypto-assets. Before engaging in high value transactions, be mindful that there may be time delays before transactions are finalized. Decentralization progress Polygon Labs is in the process of further decentralizing Polygon zkEVM. This refers to the process of gradually increasing decentralization of the system over time. - The mainnet beta will have some centralized features, such as the Sequencer and Aggregator (Prover), that Polygon Labs currently maintains in an effort to provide greater security at this time. The Sequencer has the ability to delay the inclusion of a transaction and otherwise reorder transactions. - Security of Polygon zkEVM mainnet beta is a continuous process. This process includes responding to security concerns, which depends on the Security Council. The Security Council consists of 8 individuals who are empowered to upgrade Polygon zkEVM mainnet beta without a timelock to respond to urgent security issues. If members of the Council behave maliciously or collude, then the integrity of the system may be compromised including network upgrades that may result in loss of crypto-assets. - As the Sequencer and Aggregator are centralized for mainnet beta, there are risks for potential network downtime and outages, including those that are outside the control of Polygon labs. - During the initial phase of the mainnet beta release, users may not be able to force transactions on Layer 1. Gas fees If the gas fees associated with a proposed transaction are too low, it is possible that such transaction will not be sequenced and that those fees may be lost. Security audits - Polygon Labsâ€™ implementation of Polygon zkEVM has been carefully constructed, was audited by several internal and external parties, and is continuously being reviewed and tested against engineering best practices. It is, however, unlikely that all potential bugs or vulnerabilities were identified through these audits and thus there may be undiscovered vulnerabilities that may put user funds at risk. Users should consider this risk when deciding how much value to place onto the Polygon zkEVM mainnet beta. To see the audit reports, see here. - There is a robust bug bounty program for Polygon zkEVM to help encourage the community to find critical bugs in the codebase. Head over to the Polygon zkEVM Bug Bounty page on Immunefi. Prover infrastructure - Currently the Polygon zkEVM zkProver does not run on ARM-powered Macs. For Windows users, using WSL/WSL2 is not recommended. Apple M1 chips are not supported for now, since some optimizations on the zkProver require specific Intel instructions. This means some non-M1 computers won't work regardless of the OS, for example: AMD. - In the event you are deploying a full node of Polygon zkEVM mainnet beta, be mindful that the network data is stored inside of each docker container. This means once you remove the container that network data will be lost and you will be required to resync the network data. # local-node.md: This quick start guide shows you how to deploy a zkEVM rollup stack on your local machine. It sets up and runs the following components: - Prover: zkevm-prover - zkEVM node components: zkevm-sync, zkevm-eth-tx-manager, zkevm-sequencer, zkevm-sequence-sender, zkevm-l2gaspricer, zkevm-aggregator, zkevm-json-rpc - L1 network: zkevm-mock-l1-network - zkEVM node databases: zkevm-event-db, zkevm-pool-db, zkevm-state-db - Explorer databases: zkevm-explorer-l1-db, zkevm-explorer-l2-db - Explorers: zkevm-explorer-l1, zkevm-explorer-l2, zkevm-explorer-json-rpc !!! note - The documentation describes standard deployments. - Edit the configuration files to implement your own custom setups. Prerequisites Hardware - A Linux-based OS (e.g., Ubuntu Server 22.04 LTS). - At least 16GB RAM with a 4-core CPU. - An AMD64 architecture system. !!! warning - Some of the components aren't set up to run on MacOS. Please check the workaround. The explorers and the bridge service probably won't come up. - For Windows users, WSL/WSL2 use is not recommended. !!! tip "MacOS workaround" You may see errors related to amd64 such as: sh â ‡ zkevm-prover no matching manifest for linux/arm64/v8 in the manifest list entries make: [run] Error 18 If so, go to the docker-compose.yml file in the test directory, search for the component details - in this case zkevm-prover, and add the platform: linux/amd64 environment variable under the environment variable (which you should also add if it is not there): sh environment: - EXPERIMENTALDOCKERDESKTOPFORCEQEMU=1 platform: linux/amd64 You may find the explorer components affected similarly. Software - go version 1.21 - Docker - Docker Compose 1. Clone the repo sh git clone https://github.com/0xPolygonHermez/zkevm-node.git cd zkevm-node 2. Build the Docker image bash make build-docker !!! warning - Rerun this command every time there is a code-change. 3. Test the environment zkevm-node provides commands that allow you to interact with smart contracts, run components, create encryption files, and print out debug information. !!! warning - All the data is stored inside Docker containers. - You will lose all the data whenever you remove the container. 3.1 Run the environment The test/ directory contains scripts and files for developing and debugging. bash cd test/ Then: bash make run The make run command spins up the containers that run the environment, but nothing else. This means that the L2 has no data yet. Run the explorers !!! warning The L2 explorer may not come up on MacOS regardless of environment variables. You need to bring up the explorer databases and networks separately: sh make run-explorer-db run-explorer 3.2 Stop the environment bash make stop 3.3 Restart the environment bash make restart 4. Test with sample data 4.1 Add example transactions and smart contracts This example builds and deploys contracts, funds accounts, and sends some transactions. bash make deploy-sc You should see output similar to this: sh 2024-04-17T10:05:17.729+0200 INFO deploysc/main.go:42 connecting to Local L1: http://localhost:8545 {"pid": 16869, "version": "v0.1.0"} 2024-04-17T10:05:17.729+0200 INFO deploysc/main.go:45 connected {"pid": 16869, "version": "v0.1.0"} 2024-04-17T10:05:17.735+0200 DEBUG deploysc/main.go:54 ETH Balance for 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266: 99989984676361202461 {"pid": 16869, "version": "v0.1.0"} 2024-04-17T10:05:17.735+0200 DEBUG deploysc/main.go:57 Sending TX to deploy Counter SC {"pid": 16869, "version": "v0.1.0"} 2024-04-17T10:05:19.763+0200 DEBUG operations/wait.go:101 Transaction successfully mined: 0xfac2ebc78eada5141cf3737fd5c6ad399d3d18ed7a026d60e87d00c2db0d6ed9 {"pid": 16869, "version": "v0.1.0"} ... 4.2 Deploy a full Uniswap environment This example builds out a Uniswap environment which creates token contracts and deploys them, mines tokens, and performs token swaps. bash make deploy-uniswap You should see output similar to this: sh 2024-04-17T10:07:56.298+0200 DEBUG pkg/swap.go:40 after first swap aCoin.balanceOf[0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266]: 989999999999999999950 {"pid": 17666, "version": "v0.1.0"} 2024-04-17T10:07:56.301+0200 DEBUG pkg/swap.go:43 after first swap bCoin.balanceOf[0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266]: 980000000000000000009 {"pid": 17666, "version": "v0.1.0"} 2024-04-17T10:07:56.305+0200 DEBUG pkg/swap.go:46 after first swap cCoin.balanceOf[0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266]: 990000000000000000032 {"pid": 17666, "version": "v0.1.0"} 2024-04-17T10:07:56.306+0200 DEBUG pkg/swap.go:47 Swaping tokens from B <-> C {"pid": 17666, "version": "v0.1.0"} 2024-04-17T10:07:56.307+0200 DEBUG pkg/swap.go:62 SwapExactTokensForTokens 0x70e0bA845a1A0F2DA3359C97E0285013525FFC49 <-> 0x4826533B4897376654Bb4d4AD88B7faFD0C98528 pair: 0x09dF250580cb3ba5D061f6dfcA783cDa4caa6461 {"pid": 17666, "version": "v0.1.0"} 2024-04-17T10:07:56.308+0200 DEBUG pkg/swap.go:67 SwapExactTokensForTokens 0x70e0bA845a1A0F2DA3359C97E0285013525FFC49 <-> 0x4826533B4897376654Bb4d4AD88B7faFD0C98528 reserves 0: 9999999999999999968 1: 10000000000000000036 Block Timestamp: 1713341274 {"pid": 17666, "version": "v0.1.0"} 2024-04-17T10:07:56.309+0200 DEBUG pkg/swap.go:70 SwapExactTokensForTokens 0x70e0bA845a1A0F2DA3359C97E0285013525FFC49 <-> 0x4826533B4897376654Bb4d4AD88B7faFD0C98528 exactAmountIn: 9 amountOut: 8 {"pid": 17666, "version": "v0.1.0"} 2024-04-17T10:07:58.338+0200 DEBUG operations/wait.go:101 Transaction successfully mined: 0x5c0a2fdb27f6c681b898101883f278dee1ee78dcdacb8471872e5b00603643fe {"pid": 17666, "version": "v0.1.0"} 4.3 Grant the Pol smart contract a set amount of tokens This sets up accounts with POL and is run as part of the previous examples. sh make run-approve-pol 5. Set up MetaMask To configure MetaMask to use your local environment, make sure the network is running and follow these steps. 1. Log in to your MetaMask wallet. 2. Click your account picture and then on Settings. 3. On the left menu, click Networks. Add zkEVM L2 network - Click the Add Network button and Add a network manually. - Enter the following L2 network information: 1. Network name: Polygon zkEVM - local 2. New RPC URL: 3. Chain ID: 1001 4. Currency symbol: (accept default) 5. Block explorer URL: - Click Save. Add Geth L1

network - Click the Add Network button and Add a network manually. - Enter the following L1 network information: 1. Network name: Geth - local 2. New RPC URL: 3. Chain ID: 1337 4. Currency symbol: ETH - Click Save. Environment configurations - Use the following details for the running components to set up your applications and tests. - You can find these details in the running logs also. Databases zkEVM node state database - Type: Postgres DB - User: stateuser - Password: statepassword - Database: state-db - Host: localhost - Port: 5432 - URL: zkEVM node pool database - Type: Postgres DB - User: pooluser - Password: poolpassword - Database: pooldb - Host: localhost - Port: 5433 - URL: zkEVM node JSON-RPC database - Type: Postgres DB - User: rpcuser - Password: rpcpassword - Database: rpcdb - Host: localhost - Port: 5434 - URL: Explorer L1 database - Type: Postgres DB - User: l1exploreruser - Password: l1explorerpassword - Database: l1explorerdb - Host: localhost - Port: 5435 - URL: Explorer L2 database - Type: Postgres DB - User: l2exploreruser - Password: l2explorerpassword - Database: l2explorerdb - Host: localhost - Port: 5436 - URL: Networks L1 network - Type: Geth - Host: localhost - Port: 8545 - URL: zkEVM node - Type: JSON RPC - Host: localhost - Port: 8123 - URL: Explorers Explorer L1 - Type: Web - Host: localhost - Port: 4000 - URL: Explorer L2 - Type: Web - Host: localhost - Port: 4001 - URL: Prover - Type: Mock - Host: localhost - Port: Depending on the prover image, if it's mock or not: - Prod prover: 50052 for Prover, 50061 for Merkle Tree, 50071 for Executor - Mock prover: 43061 for MT, 43071 for Executor - URL: Environment addresses The following addresses are configured into the running environment. L1 addresses | Address | Description | |---|---| | 0x8dAF17A20c9DBA35f005b6324F493785D239719d | Polygon zkEVM | | 0x40E0576c0A7dff9dc460B29ba73e79aBf73dD2a9 | Polygon bridge | | 0x5FbDB2315678afecb367f032d93F642f64180aa3 | Pol token | | 0x8A791620dd6260079BF849Dc5567aDC3F2FdC318 | Polygon GlobalExitRootManager | | 0xB7f8BC63BbcaD18155201308C8f3540b07f84F5e | Polygon RollupManager | Deployer account | Address | Private Key | |---|---| | 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266 | 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80 | Sequencer account | Address | Private Key | |---|---| | 0x617b3a3528F9cDd6630fd3301B9c8911F7Bf063D | 0x28b2b0318721be8c8339199172cd7cc8f5e273800a35616ec893083a4b32c02e | Aggregator account | Address | Private Key | |---|---| | 0x70997970C51812dc3A010C7d01b50e0d17dc79C8 | 0x59c6995e998f97a5a0044966f0945389dc9e86dae88c7a8412f4603b6b78690d | Test accounts with funds The environment also provides a bunch of test accounts that contain funds. | Address | Private Key | |---|---| | 0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC | 0x5de4111afa1a4b94908f83103eb1f1706367c2e68ca870fc3fb9a804cdab365a | | 0x90F79bf6EB2c4f870365E785982E1f101E93b906 | 0x7c852118294e51e653712a81e05800f419141751be58f605c371e15141b007a6 | | 0x15d34AAf54267DB7D7c367839AAf71A00a2C6A65 | 0x47e179ec197488593b187f80a00eb0da91f1b9d0b13f8733639f19c30a34926a | | 0x9965507D1a55bcC2695C58ba16FB37d819B0A4dc | 0x8b3a350cf5c34c9194ca85829a2df0ec3153be0318b5e2d3348e872092edffba | | 0x976EA74026E726554dB657fA54763abd0C3a0aa9 | 0x92db14e403b83dfe3df233f83dfa3a0d7096f21ca9b0d6d6b8d88b2b4ec1564e | | 0x14dC79964da2C08b23698B3D3cc7Ca32193d9955 | 0x4bbbf85ce3377467afe5d46f804f221813b2bb87f24d81f60f1fcdbf7cbf4356 | | 0x23618e81E3f5cdF7f54C3d65f7FBc0aBf5B21E8f | 0xdbda1821b80551c9d65939329250298aa3472ba22feea921c0cf5d620ea67b97 | | 0xa0Ee7A142d267C1f36714E4a8F75612F20a79720 | 0x2a871d0798f97d79848a013d4936a73bf4cc922c825d33c1cf7073dff6d409c6 | | 0xBcd4042DE499D14e55001CcbB24a551F3b954096 | 0xf214f2b2cd398c806f84e317254e0f0b801d0643303237d97a22a48e01628897 | | 0x71bE63f3384f5fb98995898A86B02Fb2426c5788 | 0x701b615bbdfb9de65240bc28bd21bbc0d996645a3dd57e7b12bc2bdf6f192c82 | | 0xFABB0ac9d68B0B445fB7357272Ff202C5651694a | 0xa267530f49f8280200edf313ee7af6b827f2a8bce2897751d06a843f644967b1 | | 0x1CBd3b2770909D4e10f157cABC84C7264073C9Ec | 0x47c99abed3324a2707c28affff1267e45918ec8c3f20b8aa892e8b065d2942dd | | 0xdF3e18d64BC6A983f673Ab319CCaE4f1a57C7097 | 0xc526ee95bf44d8fc405a158bb884d9d1238d99f0612e9f33d006bb0789009aaa | | 0xcd3B766CCDd6AE721141F452C550Ca635964ce71 | 0x8166f546bab6da521a8369cab06c5d2b9e46670292d85c875ee9ec20e84ffb61 | | 0x2546BcD3c84621e976D8185a91A922aE77ECEc30 | 0xea6c44ac03bff858b476bba40716402b03e41b8e97e276d1baec7c37d42484a0 | | 0xbDA5747bFD65F08deb54cb465eB87D40e51B197E | 0x689af8efa8c651a91ad287602527f3af2fe9f6501a7ac4b061667b5a93e037fd | | 0xdD2FD4581271e230360230F9337D5c0430Bf44C0 | 0xde9be858da4a475276426320d5e9262ecfc3ba460bfac56360bfa6c4c28b4ee0 | | 0x8626f6940E2eb28930eFb4CeF49B2d1F2C9C1199 | 0xdf57089febbacf7ba0bc227dafbffa9fc08a93fdc68e1e42411a14efcf23656e | # production-node.md: The Polygon zkEVM beta mainnet is available for developers to launch smart contracts, execute transactions, and experiment. This document shows you how to launch your own production zkNode. Developers can setup a production node with either the Polygon zkEVM mainnet or the Cardona testnet. After spinning up an instance of the production node, you can run the synchronizer and utilize the JSON-RPC interface. !!!info - Sequencer and prover functionalities are not covered in this document as they are still undergoing development and rigorous testing. - Syncing the zkNode currently takes anywhere between 1-2 days depending on various factors. The team is working on snapshots to improve the syncing time. Prerequisites This tutorial requires a docker-compose installation. Run the following to create a directory: sh mkdir -p /zkevm-node Minimum hardware requirements !!!caution - The zkProver does not work on ARM-based Macs yet. - For Windows users, the use of WSL/WSL2 is not recommended. - Currently, zkProver optimizations require CPUs that support the AVX2Â instruction, which means some non-M1 computers, such as AMD, won't work with the software regardless of the OS. - 16GB RAM - 4-core CPU - 250/350GB storage (increasing over time) Software requirements - An Ethereum node; Geth or any service providing a JSON RPC interface for accessing the L1 network. - zkEVM node (or zkNode) for the L2 network. - Synchronizer which is responsible for synchronizing data between L1 and L2. - A JSON RPC server which acts as an interface to the L2 network. Ethereum node setup We set up the Ethereum node first as it takes a long time to synchronize. We recommend using Geth but a Sepolia node is OK too. Follow the instructions provided in this guide to setup and install Geth. If you plan to have more than one zkNode in your infrastructure, we advise using a machine that is specifically dedicated to this implementation. zkNode setup Once the L1 installation is complete, we can start the zkNode setup. This is the most straightforward way to run a zkEVM node and it's fine for most use cases. However, if you want to provide service to a large number of users, you should modify the default configuration. Furthermore, this method is purely subjective and feel free to run this software in a different manner. For example, Docker is not required, you could simply use the Go binaries directly. Let's start setting up our zkNode: 1. Launch your command line/terminal and set the variables using below commands: bash define the network("mainnet" or "cardona") ZKEVMNET=cardona define installation path ZKEVMDIR=./pathtoinstall define your config directory ZKEVMCONFIGDIR=./pathtoconfig 2. Download and extract the artifacts. Note that you may need to install unzip before running this command. bash curl -L https://github.com/0xPolygonHermez/zkevm-node/releases/latest/download/$ZKEVMNET.zip > $ZKEVMNET.zip && unzip -o $ZKEVMNET.zip -d $ZKEVMDIR && rm $ZKEVMNET.zip 3. Copy the example.env file with the environment parameters: sh cp $ZKEVMDIR/$ZKEVMNET/example.env $ZKEVMCONFIGDIR/.env 4. The example.env file must be modified according to your configurations. Edit the .env file with your favorite editor (we use nano): sh nano $ZKEVMCONFIGDIR/.env bash ZKEVMNETWORK = "mainnet" or ZKEVMNETWORK = "cardona" URL of a JSON RPC for Sepolia ZKEVMNODEETHERMANURL = "http://your.L1node.url" PATH WHERE THE STATEDB POSTGRES CONTAINER WILL STORE PERSISTENT DATA ZKEVMNODESTATEDBDATADIR = "/path/to/persistent/data/stetedb" PATH WHERE THE POOLDB POSTGRES CONTAINER WILL STORE PERSISTENT DATA ZKEVMNODEPOOLDBDATADIR = "/path/to/persistent/data/pooldb" 5. To run the zkNode instance, run the following command: bash sudo docker compose --env-file $ZKEVMCONFIGDIR/.env -f $ZKEVMDIR/$ZKEVMNET/docker-compose.yml up -d 6. Run this command to check if everything went well and all the components are running properly: bash docker compose --env-file $ZKEVMCONFIGDIR/.env -f $ZKEVMDIR/$ZKEVMNET/docker-compose.yml ps You will see a list of the following containers: - zkevm-rpc - zkevm-sync - zkevm-state-db - zkevm-pool-db - zkevm-prover 7. You should now be able to run queries to the JSON-RPC endpoint at http://localhost:8545. Testing Run the following query to get the most recently synchronized L2 block; if you call it every few seconds, you should see the number grow: bash curl -H "Content-Type: application/json" -X POST --data '{"jsonrpc":"2.0","method":"ethblockNumber","params":[],"id":83}' http://localhost:8545 Stopping the zkNode Use the below command to stop the zkNode instance: bash sudo docker compose --env-file $ZKEVMCONFIGDIR/.env -f $ZKEVMDIR/$ZKEVMNET/docker-compose.yml down Updating the zkNode To update the zkNode software, repeat the setup steps, being careful not to overwrite the configuration files that you have modified. In other words, instead of running cp $ZKEVMDIR/testnet/example.env $ZKEVMCONFIGDIR/.env, check if the variables of $ZKEVMDIR/testnet/example.env have been renamed or there are new ones, and update $ZKEVMCONFIGDIR/.env accordingly. Troubleshooting - It's possible that the machine you're using already uses some of the necessary ports. In this case you can change them directly in $ZKEVMDIR/testnet/docker-compose.yml. - If one or more containers are crashing, please check the logs using the below command: bash docker compose --env-file $ZKEVMCONFIGDIR/.env -f $ZKEVMDIR/$ZKEVMNET/docker-compose.yml logs !!!info "Batch rate" - Batches are closed every 10s, or whenever they are full. - The frequency of closing batches is subject to change as it depends on the prevailing configurations. - The batch rate always needs to be updated accordingly. # configure-node-deployment.md: Set up 1. Create some working directories. !!! info The following commands take care of directory placements where non-obvious. sh mkdir -p /zkevm/data/{statedb,pooldb} /zkevm/zkevm-config /zkevm/zkevm-node 2. Populate the directories by fetching data from latest mainnet node release. sh export ZKEVMNET="mainnet" export ZKEVMDIR="zkevm" curl -L https://github.com/0xPolygonHermez/zkevm-node/releases/latest/download/$ZKEVMNET.zip > $ZKEVMNET.zip && unzip -o $ZKEVMNET.zip -d $ZKEVMDIR && rm $ZKEVMNET.zip 3. Go to the zkevm/zkevm-config directory and create an .env file and open it for editing. sh cd /zkevm/zkevm-config/ nano .env 4. In the .env file, set the following: sh ZKEVMNODEETHERMANURL = "" set valid Sepolia RPC endpoint ZKEVMNODESTATEDBDATADIR = "/zkevm/data/statedb" ZKEVMNODEPOOLDBDATADIR = "/zkevm/data/pooldb" !!! warning "Workaround if you have configuration issues" - You may have to hardcode these variables into the mainnet/docker-compose.yaml file. - Also, the ZKEVMNETWORK variable which you can set to mainnet. - Note, there is more than one reference to hardcode in. Approve POL token for sequencer 1. Launch a Hardhat console connected to the Sepolia network. sh cd /zkevm-contracts npx hardhat console --network sepolia 2. Add the missing data as directed and copy/paste the following code into the open console. js const provider = ethers.getDefaultProvider(""); // set Sepolia RPC node const privateKey = ""; // from wallets.txt const wallet = new ethers.Wallet(privateKey, provider); const polTokenFactory = await ethers.getContractFactory("ERC20PermitMock", provider); polTokenContract = polTokenFactory.attach(""); // from /zkevm-contracts/deployments/deployoutput.json polTokenContractWallet = polTokenContract.connect(wallet); await polTokenContractWallet.approve("", ethers.parseEther("100.0")); // from /zkevm-contracts/deployments/deployoutput.json Configure genesis 1. Copy the genesis.json file into the appropriate location. sh cp /zkevm-contracts/deployment/v2/genesis.json /zkevm/mainnet/config/environments/mainnet/public.genesis.config.json 2. Copy/paste the json below to the head of the public.genesis.config.json file inputting the data from /zkevm-contracts/deployments/v2/deployoutput.json. !!! important The genesisBlockNumber is called deploymentBlockNumber in deployoutput.json. json "l1Config" : { "chainId": 11155111, "polygonZkEVMAddress": "", "polTokenAddress": "", "polygonZkEVMGlobalExitRootAddress": "" }, "genesisBlockNumber": , Update node config Add the following missing parameters to the /zkevm/mainnet/config/environments/mainnet/public.node.config.toml file. !!! warning If you're having trouble locating the configuration file, try looking for node.config.toml, since it may exist under that name in some cases. Once you've added the missing parameters, rename the file to public.node.config.toml. - ApiKey for Etherscan - URL for Sepolia node, under [ETHERMAN] Add wallet keystores Copy/paste the keystore value from wallets.txt for the sequencer and aggregator respectively into the following files. sh paste only the keystore value from wallets.txt in each respective file nano /zkevm/zkevm-config/sequencer.keystore nano /zkevm/zkevm-config/aggregator.keystore # configure-prover.md: Configure prover DB 1. Copy/paste the following script into /zkevm/mainnet/db/scripts/initproverdb.sql to replace what's there. sql CREATE DATABASE proverdb; \connect proverdb; CREATE SCHEMA state; CREATE TABLE state.nodes (hash BYTEA PRIMARY KEY, data BYTEA NOT NULL); CREATE TABLE state.program (hash BYTEA PRIMARY KEY, data BYTEA NOT NULL); CREATE USER proveruser with password 'proverpass'; GRANT CONNECT ON DATABASE proverdb TO proveruser; ALTER USER proveruser SET SEARCHPATH=state; GRANT ALL PRIVILEGES ON SCHEMA state TO proveruser; GRANT ALL PRIVILEGES ON TABLE state.nodes TO proveruser; GRANT ALL PRIVILEGES ON TABLE state.program TO proveruser; 2. Save and exit the file. Configure the prover 1. Create a file called config.json and copy/paste the mock prover configuration from here: https://github.com/0xPolygonHermez/zkevm-node/blob/develop/test/config/test.prover.config.json. 2. Save it to the /zkevm/ directory. # create-wallets.md: Set up wallet contracts Clone the wallet contracts from the zkevm-contracts repository and install the npm libraries. sh git clone https://github.com/0xPolygonHermez/zkevm-contracts.git cd zkevm-contracts npm i Create wallets 1. Create a wallets.js file. sh cd zkevm-contracts nano wallets.js 2. Copy/paste the JavaScript code below. js const ethers = require("ethers"); async function main() { const arrayNames = ["Deployment Address", "\\n\\nTrusted sequencer", "\\n\\nTrusted aggregator",]; for (let i = 0; i < arrayNames.length; i++) { const wallet = ethers.Wallet.createRandom(); console.log(arrayNames[i]); console.log(Address: ${wallet.address}); console.log(PrvKey: ${wallet.privateKey}); console.log(mnemonic: "${wallet.mnemonic.phrase}"); const keystoreJson = await wallet.encrypt("password"); console.log(keystore: ${keystoreJson}); } } main().catch((e) => { console.error(e); process.exit(1); }); 3. Generate the wallets. sh node wallets.js | tee wallets.txt Prepare environment variables 1. Edit the environment variables file. bash cp .env.example .env copies .env.example file into .env nano .env opens .env file for editing 2. Set the following variables. sh MNEMONIC="..." from wallets.txt Deployment Address mnemonic INFURAAPIKEY="..." your API Key from Infura account ETHERSCANAPIKEY="..." your Etherscan API key 3. Send 1 Sepolia ETH to the deployment address wallet listed in wallets.txt. Edit deployment configuration 1. Open the deploy-parameters.json file. sh cd zkevm-contracts/deployment/v2 cp deployparameters.json.example deployparameters.json nano deployparameters.json 2. Edit the following parameters to match the generated wallet parameters. - trustedSequencer: trusted sequencer address in wallets.txt. - trustedAggregator: trusted aggregated address in wallets.txt. - admin: deployment address in wallets.txt. - zkEVMOwner: deployment address in wallets.txt. - timelockAddress: deployment address in wallets.txt. - initialZkEVMDeployerAddress: deployment address in wallets.txt. - zkEVMDeployerAddress: deployment address in wallets.txt. - emergencyCouncilAddress: deployment address in wallets.txt. - deployerPvtKey: deployment private key in wallets.txt. 3. Open the createrollupparameters.json file. bash cd zkevm-contracts/deployment/v2 cp createrollupparameters.json.example createrollupparameters.json vim createrollupparameters.json 4. Edit the following parameters to match the rollup parameters - trustedSequencer: trusted sequencer address in wallets.txt. - adminZkEVM: deployment address in wallets.txt. Deploy & verify contracts cd back to zkevm-contract root directory and run the deployment scripts. 1. Install Hardhat: cd .. npm i @openzeppelin/hardhat-upgrades 2. Deploy Polygon zkEVM deployer bash npx hardhat run deployment/v2/2deployPolygonZKEVMDeployer.ts --network sepolia You should see output that looks like this: bash ####################### polygonZkEVMDeployer deployed on: 0xB1A5BA61fBAD71Ba52d70B769d6A994c01b40983 3. Verify deployer bash npx hardhat run deployment/v2/verifyzkEVMDeployer.js --network sepolia You should see ouput that looks like this: bash The contract 0xB1A5BA61fBAD71Ba52d70B769d6A994c01b40983 has already been verified. https://sepolia.etherscan.io/address/0xB1A5BA61fBAD71Ba52d70B769d6A994c01b40983#code 4. Prepare testnet bash npx hardhat run deployment/testnet/prepareTestnet.ts --network sepolia You should see output that looks like this: bash ####################### pol deployed to: 0x2B2Ef864542EA38657221393B0A18215e5c3fc7e And now if you go to sepolia scan, you should also see that under your account, there's a new Pol Erc-20 token created with the balance of 19,900,000 tokens. # environment-overview.md: Deploying the full zkEVM stack requires installing and deploying a number of different components. !!! info Each component runs in its own Docker container. | Component | Container | Brief\ Description | | :--------------- | :------------------ | -- | | Sequencer | zkevm-sequencer | Fetches txs from the pool DB, checks if valid, then puts valid ones into a batch. | | Aggregator | zkevm-aggregator | Validates sequenced batches by generating verifiable zero-knowledge proofs. | | Synchronizer | zkevm-sync | Updates the state by fetching data from Ethereum through the Etherman. | | JSON RPC | zkevm-rpc | An interface for interacting with the network. e.g., Metamask, Etherscan or Bridge. | | State DB | zkevm-state-db | A database for permanently storing state data (apart from the Merkle tree). | | Prover | zkevm-prover-server | Used by the aggregator to create zk-proofs. The full prover is extremely resource-heavy and runs on an external cloud server. Use the mock prover for evaluation/test purposes. | | Pool DB | zkevm-pool-db | Stores txs from the RPC nodes, waiting to be put in a batch by the sequencer. | | Executor | zkevm-executor | Executes all processes. Collects resultsâ€™ metadata (state root, receipts, logs) | | Etherman | zkevm-eth-tx-manager | Implements methods for all interactions with the L1 network and smart contracts. | | Bridge UI | zkevm-bridge-ui | User-interface for bridging ERC-20 tokens between L2 and L1 or another L2. | | Bridge DB | zkevm-bridge-db | A database for storing bridge-related transactions data. | | Bridge service | zkevm-bridge-service | A backend service enabling clients like the web UI to interact with bridge smart contracts. | | zkEVM explorer | zkevm-explorer-l2 | L2 network's block explorer. i.e., The zkRollup Etherscan explorer. | | zkEVM explorer DB | zkevm-explorer-l2-db | Database for the L2 network's Block explorer. i.e., Where all the zkRollup Etherscan explorer queries are made. | | Gas pricer | zkevm-l2gaspricer | Responsible for suggesting the gas price for the L2 network fees. | | Sepolia execution | sepolia-execution | L1 node's execution layer. | | Sepolia consensus | sepolia-consensus | L1 node's consensus layer. # install-dependencies.md: Open a terminal window and run the following commands to install the required software. Install base dependencies sh sudo apt update -y sudo apt install -y tmux git curl unzip jq aria2 pv Install Docker sh sudo apt-get install docker-ce sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin sudo usermod -aG docker $USER newgrp docker && newgrp $USER Install Node/npm sh curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/master/install.sh | bash source /.bashrc nvm install 18 node -v Install Golang sh wget https://go.dev/dl/go1.20.4.linux-amd64.tar.gz sudo rm -rf /usr/local/go && sudo tar -C /usr/local -xzf go1.20.4.linux-amd64.tar.gz rm -rf go1.20.4.linux-amd64.tar.gz Confirm the Golang installation with $ go version. # intro.md: This tutorial takes you through the process of deploying a full zkEVM stack on either testnet or mainnet. It relies on specific component versions so we have hidden most of the configuration complexity in scripts to make the process straightforward. !!! warning - The tutorial is for learning purposes only and is not intended to describe a latest stable version build. - Report any content issues on our docs repo: https://github.com/0xPolygon/polygon-docs Step-by-step The process is split into the following sections: 1. Overview of the full environment. 2. Prerequisite steps: Preliminary setup, checking system requirements, and prerequisite variables. 3. Install dependencies: Install required dependencies and downloading mainnet files. 4. Create wallets: Create wallets and deploy contracts. 5. Configure node deployment: Deploy the zkRollup EVM-compatible node. 6. Configure prover: Configure the prover database and the prover service. 7. Start node: Start the node. 8. Start services: Start the services. # prerequisites.md: Hardware - A Linux-based OS (e.g., Ubuntu Server 22.04 LTS). - At least 16GB RAM with a 4-core CPU. - An AMD64 architecture system. !!! info "Computing resources" - Running a full prover is an enterprise level installation and extremely resource-heavy. - These instructions run the mock prover instead. Software The commands on the install dependencies page fulfill all software requirements. Mock prover requirements !!! info The mock prover is a light resource by skipping validation and instead adding a Valid âœ… checkmark to every batch. - 4-core CPU - 8GB RAM (16GB recommended) Miscellaneous requirements - INFURAAPIKEY: Infura API key - ETHERSCANAPIKEY: Etherscan API key - Sepolia node RPC URL: e.g. https://sepolia.infura.io/v3/YOUR-INFURA-API-KEY - Sepolia account address holding minimum 0.5 SepoliaETH # start-node.md: Start node From the zkevm root directory, run the following: sh sudo docker compose -f mainnet/docker-compose.yml up This command spins up the following services: - RPC node - Synchronizer - State DB - Pool DB - Mock prover Log sample ??? "Logs sample" zkevm-rpc | /go/pkg/mod/github.com/urfave/cli/v2@v2.25.7/command.go:274 zkevm-rpc | github.com/urfave/cli/v2.(Command).Run zkevm-rpc | /go/pkg/mod/github.com/urfave/cli/v2@v2.25.7/command.go:267 zkevm-rpc | github.com/urfave/cli/v2.(App).RunContext zkevm-rpc | /go/pkg/mod/github.com/urfave/cli/v2@v2.25.7/app.go:332 zkevm-rpc | github.com/urfave/cli/v2.(App).Run zkevm-rpc | /go/pkg/mod/github.com/urfave/cli/v2@v2.25.7/app.go:309 zkevm-rpc | main.main zkevm-rpc |

/src/cmd/main.go:191 zkevm-rpc | runtime.main zkevm-rpc | /usr/local/go/src/runtime/proc.go:267 zkevm-rpc | 2024-01-24T11:32:03.043Z INFO config/config.go:163 config file not found {"pid": 1, "version": "v0.4.4"} zkevm-rpc | Version: v0.4.4 zkevm-rpc | 2024-01-24T11:32:03.045Z INFO cmd/run.go:52 Starting application {"pid": 1, "version": "v0.4.4"} zkevm-rpc | Git revision: 9ef6f20 zkevm-rpc | Git branch: HEAD zkevm-rpc | Go version: go1.21.5 zkevm-rpc | Built: Tue, 12 Dec 2023 17:18:45 +0000 zkevm-rpc | OS/Arch: linux/amd64 zkevm-rpc | 2024-01-24T11:32:03.054Z ERROR db/db.go:117 error getting migrations count: ERROR: relation "public.gorpmigrations" does not exist (SQLSTATE 42P01) zkevm-rpc | /src/log/log.go:142 github.com/0xPolygonHermez/zkevm-node/log.appendStackTraceMaybeArgs() zkevm-rpc | /src/log/log.go:217 github.com/0xPolygonHermez/zkevm-node/log.Error() zkevm-rpc | /src/db/db.go:117 github.com/0xPolygonHermez/zkevm-node/db.checkMigrations() zkevm-rpc | /src/db/db.go:53 github.com/0xPolygonHermez/zkevm-node/db.CheckMigrations() zkevm-rpc | /src/cmd/run.go:263 main.checkStateMigrations() zkevm-rpc | /src/cmd/run.go:70 main.start() zkevm-state-db | 2024-01-24 13:49:21.909 UTC [78] ERROR: relation "public.gorpmigrations" does not exist at character 22 Troubleshooting Configuration issues If you have errors related to configuration issues, see the warning at step 4 in the configure node deployment section. Process binding issue If you need to restart, make sure you kill any hanging db processes with the following commands. !!! info You can find the port number from the log warnings. sh sudo lsof -t -i: kill -9 Kill all Docker containers and images You can fix many restart issues and persistent errors by stopping and deleting all Docker containers and images. sh docker rm $(docker ps -aq) docker rmi $(docker images -q) # start-services.md: !!! warning - Please come back soon to see the updated documentation. Start L2 gas pricer !!! info Docs in progress. Start transaction manager !!! info Docs in progress. Start sequencer !!! info Docs in progress. Start aggregator !!! info Docs in progress. Start block explorer !!! info Docs in progress. Start the bridge !!! info Docs in progress. # using-foundry.md: Any smart contract deployable to the Ethereum network can be deployed easily to the Polygon zkEVM network. In this guide, we demonstrate how to deploy an ERC-721 token contract on the Polygon zkEVM network using Foundry. We follow the Soulbound NFT tutorial from this video. Set up the environment Foundry is a smart contract development toolchain. It can be used to manage dependencies, compile a project, run tests and deploy smart contracts. It also lets one interact with the blockchain from the CLI or via Solidity scripts. Install Foundry If you have not installed Foundry, Go to book.getfoundry and select Installation from the side menu. Follow the instructions to download Using Foundryup. Next, select Creating a New Project from the sidebar. Initialize and give your new project a name: forge init zkevm-sbt In case of a library not loaded error, you should run below command and then repeat the above process again: bash brew install libusb If you never installed Rust or need an update, visit the website here. Build a project and test Run the command forge build to build the project. The output should look something like this: !Successful forge build command Now, test the build with forge test !Testing Forge Build You can check out the contents of the newly built project by switching to your IDE. In case of VSCode, just type: code . Writing the smart contract 1. Find the OpenZeppelin Wizard in your browser, and use the wizard to create an out-of-the-box NFT contract. - Select the ERC721 tab for an NFT smart contract. - Name the NFT and give it an appropriate symbol. Example: Name SoEarly and Symbol SOE. - Go ahead and select features for your token. Simply tick the relevant boxes. - You can tick the URI Storage box if you wish to attach some image or special text to the token. 2. Open your CLI and install dependencies with this command: bash npm install @openzeppelin/contracts-upgradeable 3. Remap dependencies to easy-to-read filenames with the command: bash forge remappings > remappings.txt 4. Inside the new remapping.txt file, rename the referencing openzeppelin-contracts to openzeppelin, which is the name used when importing. That is, change openzeppelin-contracts/=lib/openzeppelin-contracts → openzeppelin/=lib/openzeppelin-contracts. 5. Copy the smart contract code in OpenZeppelin: Copy to Clipboard 6. In the IDE, open a new .sol file, name it and paste the copied code to this file. This is in fact the actual smart contract for the NFT. Add control on token transfers The aim here is to put rules in place stipulating that the token cannot be transferred without burning it. - Go to the OpenZeppelin documentation. - Look up the signature by searching for beforetokentransfererc721. - Scroll down to ERC 721 and copy the corresponding text on the right side: c beforeTokenTransfer(address from, address to, uint256 firstTokenId, uint256 batchSize) internal - Create a new function in the code for the smart contract token called beforeTokenTransfer c function beforeTokenTransfer (address from, address to, uint256 firstTokenId, uint256 batchSize) internal override { require(from==address(0) || to==address(0), "Soulbound: cannot transfer"); super.beforeTokenTransfer(from, to, firstTokenId, batchSize); } Set a token URI (optional) A token URI is a function that returns the address where the metadata of a specific token is stored. The metadata is a . json file where all the data associated with the NFT is stored. Our aim here is to attach some image to the created token. The stored data typically consists of the name of the token, brief description and URL where the image is stored. - Choose an image and give it a name relatable to the token - Find an IPFS storage service for the image, for example filebase or web3.storage that provide free options, or NFT.storage flagship product that charge a one-time fee per GB of storage. - Upload the image to the storage using your GitHub account Add URI json file This is the file that contains the metadata for the token which includes the image address (i.e., the IPFS address of the image). - In the IDE, create a new .json file which you can call tokenuri.json - Populate the tokenuri.json file with the token-name, description and URL where the image is stored: json { "title": "So Early", "description": "I was super duper early to the Polygon zkEVM", "image": "" / remove the forward-slash at the end of the URL, if any / } - Upload the tokenuri.json file to the same storage where the image was uploaded - Copy the address to the Sbt.sol inside the safeMint function - Remove the uri parameter so as to hardcode it. This results in all minted tokens sharing the same uri image, but each token's tokenId differs from the previous one by 1. Populate the .env file In order to deploy on the zkEVM Testnet, populate the .env file in the usual way. That is, - Create a .env.sample file within the src folder - Populate .env.sample file with your ACCOUNTPRIVATEKEY and the zkEVM Testnet's RPC URL found here. So the .env.sample file looks like this: json RPCURL="https://rpc.cardona.zkevm-rpc.com" PVTKEY="" - Copy the contents of the .env.sample file to the .env file, bash cp .env.sample .env !!!warning Make sure .env is in the .gitignore file to avoid uploading your ACCOUNTPRIVATEKEY. Deploy your contract 1. In the CLI, use the following command to ensure grabbing variables from .env: bash source .env 2. Check if the correct RPC URL is read from the .env file: bash echo $RPCURL 3. You can now use the next command: bash forge create --rpc-url $RPCURL --private-key $PRIVATEKEY src/{ContractFile.sol}:{ContractName} --legacy which executes the following: - Does a forge create. - Passes the RPCURL and PVTKEY. - References the actual smart contract. For example, when deploying the Sbt.sol contract, the command looks like this: bash forge create --rpc-url $RPCURL --private-key $PRIVATEKEY src/Sbt.sol:SoEarly --legacy The above command compiles and deploys the contract to the zkEVM Testnet. The output on the CLI looks like this one below. !Successful Deploy Sbt.sol Check deployed contract in explorer - Copy the address of your newly deployed contract (i.e. the Deployed to: address as in the above example output). - Go to the zkEVM Testnet Explorer, and paste the address in the Search by address field. - Check Transaction Details reflecting the From address, which is the owner's address and the To address, which is the same Deployed to: address seen in the CLI. # using-hardhat.md: Hardhat is one of the popular smart contract development frameworks. It is the Polygon zkEVM's preferred framework, and therefore used in the zkEVM as a default for deploying and automatically verifying smart contracts. This document is a guide on how to deploy a smart contract on the Polygon zkEVM network using Hardhat. Feel free to check out the tutorial video available here. Initial setup !!!info Before starting with this deployment, please ensure that your wallet is connected to the Polygon zkEVM Testnet. See the demo here for details on how to connect your wallet. - Add the Polygon zkEVM Testnet to your Metamask wallet and get some Testnet ETH from the Polygon Faucet. - Clone the repo using below command: bash git clone https://github.com/oceans404/fullstack-zkevm - Install dependencies and start React app (you can copy all three lines in one go). bash cd fullstack-zkevm npm i npm start Correct installation opens up the Counter App at localhost:3000. You can test it by clicking on the +1 button several times. - Back in the CLI, install dependencies: bash npm install ethers hardhat @nomiclabs/hardhat-waffle ethereum-waffle chai @nomiclabs/hardhat-ethers dotenv - Populate the .env.sample file with your ACCOUNTPRIVATEKEY ??? "How to get your private key in MetaMask" Click the vertical 3 dots in the upper-right corner of Metamask window. Select Account details and then click Export private key. Enter your Metamask password to reveal the private key. Copy the private key and paste it into the .env.sample file. - Copy the contents of the .env.sample file to the .env file, bash cp .env.sample .env Hardhat smart contract Next is the initialization of a project using Hardhat. Hardhat cannot initialize a sample project if there is an existing README file. To avoid clashes, rename any existing README.md temporarily before initializing Hardhat. bash mv README.md README-tutorial.md - Initialize a project with Hardhat: npx hardhat init. - Next, (... To avoid failure ... please go slow with this cli dialogue...), The aim here is to achieve the following outcome: !Figure So then, - Press to set the project root. - Press again to accept addition of .gitignore. - Type n to reject installing sample project's dependencies. The idea here is to postpone installing dependencies to later steps due to a possible version-related bug. - Open the hardhat.config.js file and paste the below code: js require("dotenv").config(); require("@nomicfoundation/hardhat-toolbox"); / @type import('hardhat/config').HardhatUserConfig / module.exports = { solidity: "0.8.9", paths: { artifacts: "./src", }, networks: { zkEVM: { url: "https://rpc.cardona.zkevm-rpc.com", accounts: [process.env.ACCOUNTPRIVATEKEY], }, }, }; Note that a different path to artifacts is added so that the React app can read the contract ABI within the src folder. Add scripts - Create a new file, in the contracts folder, named Counter.sol: touch contracts/Counter.sol. - Copy the below code and paste it in the Counter contract code: solidity //SPDX-License-Identifier: MIT pragma solidity ^0.8.9; contract Counter { uint256 currentCount = 0; function increment() public { currentCount = currentCount + 1; } function retrieve() public view returns (uint256){ return currentCount; } } - Create a new file in the scripts folder deploy-counter.js: touch scripts/deploy-counter.js. - Add the code below to the deploy-counter.js file: js const hre = require("hardhat"); async function main() { const deployedContract = await hre.ethers.deployContract("Counter"); await deployedContract.waitForDeployment(); console.log(Counter contract deployed to https://cardona-zkevm.polygonscan.com/address/${deployedContract.target}); } main().catch((error) => { console.error(error); process.exitCode = 1; }); - Before compiling the contract, you need to install the toolbox. You may need to change directory to install outside the project. Use this command: bash npm install --save-dev @nomicfoundation/hardhat-toolbox - Compile your contract code (i.e., go back to the project root in the CLI): bash npx hardhat compile - Now run the scripts: bash npx hardhat run scripts/deploy-counter.js --network zkEVM â€‹Here's an output example: Counter contract deployed to https://cardona-zkevm.polygonscan.com/address/0x5FbDB2315678afecb367f032d93F642f64180aa3 Update frontend The next step is to turn Counter.sol into a dApp by importing the ethers and the Counter file, as well as logging the contract's ABI. - Include the below code in the App.js file: js import { ethers } from "ethers"; import Counter from "./contracts/Counter.sol/Counter.json"; const counterAddress = "your-contract-address" console.log(counterAddress, "Counter ABI: ", Counter.abi); - Update the counterAddress to your deployed address. - It is the hexadecimal number found at the tail-end of the output of the last npx hardhat run ... command and looks like this 0x5FbDB2315678afecb367f032d93F642f64180aa3. - It must be pasted in the App.js to replace your-contract-address. Be sure to use the deployed address from your own implementation! - Update frontend counter to read from blockchain. Include the below code in the App.js file: js useEffect(() => { // declare the data fetching function const fetchCount = async () => { const data = await readCounterValue(); return data; }; fetchCount().catch(console.error); }, []); async function readCounterValue() { if (typeof window.ethereum !== "undefined") { const provider = new ethers.providers.Web3Provider(window.ethereum); console.log("provider", provider); const contract = new ethers.Contract(counterAddress, Counter.abi, provider); console.log("contract", contract); try { const data = await contract.retrieve(); console.log(data); console.log("data: ", parseInt(data.toString())); setCount(parseInt(data.toString())); } catch (err) { console.log("Error: ", err); alert("Switch your MetaMask network to Polygon zkEVM Testnet and refresh this page!"); } } } - Also, to import useEffect, insert it like this: js import { useState, useEffect } from "react"; - To be able to track a loader, add this to your state: js const [isLoading, setIsLoading] = useState(false); - This is within the App() function. - Let frontend counter write to the blockchain by adding the below requestAccount and updateCounter functions: js async function requestAccount() { await window.ethereum.request({ method: "ethrequestAccounts" }); } async function updateCounter() { if (typeof window.ethereum !== "undefined") { await requestAccount(); const provider = new ethers.providers.Web3Provider(window.ethereum); console.log({ provider }); const signer = provider.getSigner(); const contract = new ethers.Contract(counterAddress, Counter.abi, signer); const transaction = await contract.increment(); setIsLoading(true); await transaction.wait(); setIsLoading(false); readCounterValue(); } } Place these two functions above the readCounterValue() function in the App.js file. - Replace the incrementCounter function with this one: js const incrementCounter = async () => { await updateCounter(); }; - Update the increment button code to: js {isLoading ? "loading..." : "+1"} Now, run the Counter dApp by simply using npm start in CLI at the project root. Congratulations for reaching this far. You have successfully deployed a dApp on the Polygon zkEVM testnet. # verify-contract.md: Once a smart contract is deployed to zkEVM, it can be verified in various ways depending on the framework of deployment as well as the complexity of the contract. The aim here is to use examples to illustrate how you can manually verify a deployed smart contract. Ensure that your wallet is connected while following this guide. Although any wallet can be used, we use Metamask wallet throughout this tutorial. Manual verification After successfully compiling a smart contract, follow the next steps to verify your smart contract. 1. Copy the address to which the smart contract is deployed. 2. Navigate to the zkEVM Explorer and paste the contract address into the Search box. This opens a window with a box labelled Contract Address Details. 3. Scroll down to the box with tabs labelled Transactions, Internal Transactions, Coin Balance History, Logs, and Code. 4. Click the Transaction Hash in the Contract Creation box, which is the super long number. 5. Select the Code tab. 6. Click the Verify and Publish button. 7. There are 3 options to provide the Contract's code. We dive into the Flattened source code and Standard input JSON options. ??? "Flattened source code" Click Next after selecting the via Flattened Source Code option. Various frameworks have specific ways to flatten the source code. Our examples are Remix and Foundry. Using Remix In order to flatten the contract code with Remix, one needs to only right-click on the contract name and select Flatten option from the drop-down menu that appears. See the below figure for reference. !Selecting the flatten code option After selecting Flatten, a new .sol file with the suffix flatten.sol is automatically created. Copy the contents of the new flatten.sol file and paste into the Enter the Solidity Contract field in the explorer. Using Foundry In order to flatten the contract code using Foundry, the following command can be used: bash forge flatten src/ -o .sol With this command, the flattened code gets saved in the .sol file. Copy the contents of the new.sol file and paste into the Enter the Solodity Contract field in the explorer. ??? "Standard input JSON" Click Next after selecting the via Standard Input JSON option. 1. In order to update the Compiler based on your contract's compiler version, - Click the ↓ for a list of compiler versions. - Select the corresponding version. For example, select v0.8.9+commit.e5eed63a if your code has pragma solidity ^0.8.9;. 2. Paste the Standard input JSON file into the Drop the standard input JSON file or Click here field. You can find it in your local project folder. - The Standard input JSON file is the {"superlongnumberfile"}.json in the build-info subfolder. Path example: fullstack-zkevm/src/build-info/{"superlongnumberfile"}.json - Save this file to parse it with Prettier - Find the input JSON object. It looks something like this → "input": {} - Copy the input object value into a new file - Name and save this file locally in the root folder. Check this example file for reference - Drag and drop the Standard Input JSON file into Drop the standard input JSON file or Click here field. Once pasted, the Verify & Publish button becomes active. 3. Since you have provided an input, set Try to fetch constructor arguments automatically to No. 4. To add your ABI-encoded constructor arguments: - Open the Online ABI Encoder - Choose the Auto-parse tab. - Copy the ABI-encoded output. - Paste it into ABI-encoded Constructor Arguments if required by the contract. Once you paste the contents of the newly created .sol file to the Enter the Solidity Contract field, the Verify & Publish button becomes active. Click on Verify & Publish to verify your deployed smart contract. Verify using Remix We use the ready-made Storage.sol contract in Remix. Compile the contract and follow the steps provided below. 1. Deploy the Storage.sol contract: - Click the Deploy icon on the left-side of the IDE window. - Change ENVIRONMENT to "Injected Provider - MetaMask" (ensure that your wallet is already connected to Sepolia network). - Confirm the connection request when MetaMask pops up. - Click the Deploy button and confirm. 2. Check the deployed smart contract on Etherscan: - Copy the contract address below the Deploy Contracts. - Navigate to the Sepolia explorer. - Paste the contract address in the Search by address field and press ENTER. - Click on the Transaction Hash to see transaction details. 3. You are going to need your Etherscan API Key in order to verify. - Login to Etherscan. - Hover the cursor over your username for a drop-down menu. - Select API Keys. - Click API Keys again below the Others option. - Copy the API Key. 4. Next, in the Remix IDE: - Click Plugin Manager icon on the bottom-left corner of the Remix IDE. - Type Etherscan in the search field at the top. - Click Activate button as the Etherscan option appears. An Etherscan icon should appear on the left-side of the IDE. - Click on the Etherscan icon. - Ensure that Sepolia is present in the Selected Network field. - Click within the Contract Name field and type in the name of your deployed contract, or select it if it appears. - Paste the address in the Contract Address field. - Verify button becomes active if all information has been provided. - Click the Verify button to complete verification of your smart contract. # write-contract.md: This document explains how to automatically write a smart contract using the OpenZeppelin Wizard. The resulting smart contract code can either be integrated with Remix by Clicking the Open in Remix button, or copied to a clipboard and pasted in the user's intended IDE. Getting started Navigate to the OpenZeppelin Wizard in your browser. First thing to notice is the Solidity Wizard and Cairo Wizard buttons. One can choose any of the following tabs to begin creating an out-of-box smart contract code in either Solidity (for EVM chains) or Cairo (useful for Starknet). These are: - ERC20 for writing an ERC-20 token smart contract. - ERC721 for writing an NFT token smart contract. - ERC1155 for writing an ERC-1155 token smart contract. - Governor for creating a DAO. - Custom for writing a customized smart contract. Writing an NFT contract For illustration purposes, we will be creating a NFT smart contract. Suppose you wanted to create a Mintable, Burnable ERC721 token and specify an appropriate license for it. 1. Select the ERC721 tab. 2. Give your NFT a name and a symbol by filling the Name and Symbol fields. 3. Use the check-boxes on the left to select features of your token. - Put a tick on the Mintable check-box. - Put a tick on the Auto Increment Ids check-box, this ensures uniqueness of each minted NFT. - Put a tick on the Burnable check-box. - Either leave the default MIT license or type the license of your choice. Notice that new lines of code are automatically written each time a feature is selected. Voila! Contract is ready With the resulting lines of code, you now have the NFT token contract written in Solidity. As mentioned above, this source code can now be ported to an IDE of your choice or opened directly in Remix. The below figure depicts the auto-written NFT smart contract code. !The end-product NFT source code # evm-differences.md: This document provides brief remarks on the differences between the EVM and the Polygon zkEVM. Lists of supported and unsupported EIPs, opcodes, and additional changes made when building the Polygon zkEVM, can be found here. EVM-equivalence Polygon zkEVM is designed to be EVM-equivalent rather than just compatible. The difference between EVM-compatibility and EVM-equivalence is that; - Solutions that are compatible support most of the existing applications, but sometimes with code changes. Additionally, compatibility may lead to breaking developer tooling. - Polygon zkEVM strives for EVM-equivalence which means most applications, tools, and infrastructure built on Ethereum can immediately port over to Polygon zkEVM, with limited to no changes needed. Things are designed to work 100% on day one. EVM-equivalence is critical to Polygon zkEVM for several reasons, including the following: 1. Development teams don't have to make changes to their code, and this eliminates the possibility of introducing new security vulnerabilities. 2. No code changes means no need for additional audits. This saves time and money. 3. Since consolidation of batches and finality of transactions is achieved via smart contracts on Ethereum, Polygon zkEVM benefits from the security of Ethereum. 4. EVM-equivalence allows Polygon zkEVM to benefit from the already vibrant and active Ethereum community. 5. It also allows for significant and quick dApp adoption, because applications built on Ethereum are automatically compatible. Ultimately, Polygon zkEVM offers developers the same UX as on Ethereum, with significantly improved scalability. The following differences have no impact on the developer's experience on the zkEVM compared to the EVM: - Gas optimization techniques. - Interacting with libraries, like Web3.js and Ethers.js. - Deploying contracts seamlessly on the zkEVM without any overhead. # index.md: This section contains specifications of other zkEVM features not covered in the Architecture section of this documentation. First are the two novel languages: The zero-knowledge Assembly (zkASM) language which interprets the firmware of microprocessor-type state machines, and the polynomial identity language (PIL) which is instrumental in enabling verification of state transitions. Second are some of the differences between Polygon zkEVM and the EVM. These are differences in terms of opcodes, supported precompiled contracts, newly added features and other variances. # compiling-using-pilcom.md: This document describes how Polynomial Identity Language programs are compiled by PILCOM. Depending on the language used in implementation, every PIL code can be compiled into either a $\texttt{JSON}$ file or a $\texttt{C++}$ code by using a compiler called pilcom. The pilcom compiler package can be found at this Github repository here. Setup can be fired up at the command line with the usual $\texttt{clone}$, $\texttt{install}$ and $\texttt{build}$ CLI commands. Any PIL code can be compiled into a $\texttt{JSON}$ file with the command, bash node src/pil.js -o which is a basic $\texttt{JSON}$ representation of the PIL program (with some extra metadata) to be later consumed on by the pil-stark package in order to generate a STARK proof. Similarly, any PIL code can be compiled into C++ code with this command, bash node src/pil.js -c -n namespace in which case the corresponding header files (.hpp) will be generated in the ./polsgenerated folder. Restriction on polynomial degrees The current version of PIL can only handle quadratics. That is, given any set of polynomials; $\texttt{a}$, $\texttt{b}$ and $\texttt{c}$; PIL can only handle products of two polynomials at a time, $$ \mathtt{a a},\ \ \mathtt{a b}\ \ \text{and}\ \ \mathtt{a c} $$ but not higher degrees such as, $$ \mathtt{a a a},\ \ \mathtt{a b b},\ \ \mathtt{(1-a) b c}\ \ \text{ or }\ \ \mathtt{a b c c} $$ These higher degree products are handled via an $\texttt{intermediate}$ polynomial, conveniently dubbed $\texttt{carry}$. Consider again the constraint of the optimized Multiplier program: $$ \texttt{out}' = \texttt{RESET} \texttt{freeIn}\ +\ (1 - \texttt{RESET}) (\texttt{freeIn} \texttt{out}) \tag{Eqn. 6} $$ which involves the trinomial, $$ (1 - \texttt{RESET}) (\texttt{freeIn} \texttt{out}) . $$ A $\texttt{carry}$ can be used in Eqn. 6 as follows, $$ \texttt{out}' = \texttt{RESET} \texttt{freeIn}\ +\ (1 - \texttt{RESET}) \texttt{carry} \tag{Eqn. 7} $$ where in this case, $\texttt{carry} = \texttt{freeIn} \texttt{out}$. In the same sense that keywords $\texttt{commit}$ and $\texttt{constant}$ can be thought of as types of polynomials, $\texttt{intermediate}$ can also be regarded as a third type of polynomial in PIL. PIL compilation In order to compile the above PIL code to a JSON file, follow the following steps. - Create a subdirectory/folder for the Multiplier SM and call it multipliersm. - Switch directory to the new subdirectory multipliersm, and open a new file. Name it multiplier.pil , copy in it the text below and save; namespace Multiplier(210); // Constant Polynomials pol constant RESET; // Committed Polynomials pol commit freeIn; pol commit out; // Intermediate Polynomials pol carry = outfreeIn; // Constraints out' = RESETfreeIn + (1-RESET)carry; - Switch directory to $\texttt{pilcom}/$ and run the below command, bash node src/pil.js /multipliersm/multiplier.pil -o multiplier-1st.json If compilation is successful, the following debug message will be printed on the command line, Input Pol Commitments: 2 Q Pol Commitmets: 1 Constant Pols: 1 Im Pols: 1 plookupIdentities: 0 permutationIdentities: 0 connectionIdentities: 0 polIdentities: 1 The debug message reflects the numbers of; - Input committed polynomials, denoted by $\texttt{Input Pol Commitments}$. - Quadratic polynomials, denoted by $\texttt{Q Pol Commitmets}$. - Constant polynomials, denoted by $\texttt{Constant Pols}$. - Intermediate polynomials, denoted by $\texttt{Im Pols}$. - The various identities that can be checked; the $\texttt{Plookup}$, the $\texttt{Permutation}$, the $\texttt{connection}$ and the $\texttt{Polynomial}$ identities. The resulting $\texttt{JSON}$ file into which the multiplier.pil code is compiled looks like this: json { "name": "multipliersm", "version": "1.0.0", "description": "", "main": "index.js", "scripts": { "test": "echo \"Error: no test specified\" && exit 1" }, "keywords": [], "author": "", "license": "ISC" } This $\texttt{JSON}$ file contains all the information needed by the proof/verification package called $\texttt{pil-stark}$ for processing. # configuration-files.md: This document describes why Polynomial Identity Language uses a special configuration file. In order for PIL to securely enable modularity, especially in complex settings such as the Polygon zkEVM's, where the Main SM has several secondary state machines executing different computations, a dependency inclusion feature among different .pil files needed to be developed. Dependency inclusion feature Let's consider a scenario. If the PIL code of Secondary SMs reflects unique properties such as the maximum length (for example, the length $\mathtt{2^{10}}$ of the Multiplier SM as seen in the first line of the multiplier.pil code), such properties can easily become magic numbers which attackers could use as distinguishers of which computation is running at a given point in time. This is where the dependency inclusion feature comes in. In order to circumvent such possible attacks, a common configuration file written in PIL called config.pil, is created to contain configuration-related properties shared among various programs. Therefore, the file config.pil gets included in the PIL codes of relevant programs, and constants are no longer declared by their values but with keywords or symbols. Below is the PIL code of the Optimized Multiplier SM, with the config.pil file. include "config.pil"; namespace Multiplier(%N); // Constant Polynomials pol constant RESET; // Committed Polynomials pol commit freeIn; pol commit out; // Intermediate Polynomials pol carry = outfreeIn; // Constraints out' = RESETfreeIn + (1-RESET)carry; Observe that the number $\mathtt{2^{10}}$ does not appear in the PIL code but the symbol "$\texttt{%N}$". In this particular example, it means the config.pil file contains the value $\mathtt{2^{10}}$ as indicated below. constant %N = 210; The compiler distinguishes between constant's identifiers and other identifiers via the percent symbol (%). Therefore, all constant identifiers in PIL should be preceded by the percent symbol (%). # connection-arguments.md: This document describes the connection arguments and how they are used in Polynomial Identity Language. What is a connection argument? Given a vector $a = (a1 , \dots , an) \in \mathbb{F}n$ and a partition ${\large{\S}} = \{ S1, \dots , St \}$ of $[n]$, we say "a $\textit{copy-satisfies}$ ${\large{\S}}$" if for each $Sk \in {\large{\S}}$, we have that $ai = aj$ whenever $i, j \in Sk$, with $i, j \in [n]$ and $k \in [t]$. Moreover, we say that a protocol $(\mathcal{P}, \mathcal{V})$ is a connection argument if the protocol can be used by $\mathcal{P}$ to prove to $\mathcal{V}$ that a vector $\textit{copy-satisfies}$ a partition of $[n]$. !!! info We use the term â€œconnectionâ€ instead of â€œcopy-satisfactionâ€ because the argument is used in PIL in a more general sense than in the original definition given in GWC19. Example Let ${\large{\S}} = \{\{2\}, \{1, 3, 5\}, \{4, 6\}\}$ be a specified partition of $[6]$. Observe the two columns depicted below: $$ \begin{aligned} \begin{array}{|c|c|}\hline \texttt{ a }\\ \hline \text{3}\\ \text{9}\\ \text{3}\\ \text{1}\\ \text{3}\\ \text{1}\\ \hline \end{array} \end{aligned} \hspace{0.2cm} \begin{aligned} \begin{array}{|c|c|}\hline \texttt{ b }\\ \hline \text{3}\\ \text{9}\\ \text{7}\\ \text{1}\\ \text{3}\\ \text{1}\\ \hline \end{array} \end{aligned} \tag{Table 1} $$ The vector $\mathtt{a}\ \textit{copy-satisfies}\ {\large{\S}}$ because $\mathtt{a}1 = \mathtt{a}3 = \mathtt{a}5 = 3$ and $\mathtt{a}4 = \mathtt{a}6 = 1$. Observe that, since the singleton $\{2\}$ is in ${\large{\S}}$, then $\mathtt{a}2$ is not related to any other element in $\mathtt{a}$. Also, the vector $\mathtt{b}$ does not $\textit{copy-satisfies}\ {\large{\S}}$ because $\mathtt{b}1 = \mathtt{b}5 =3 \not= 7 = \mathtt{b}3$. In the context of programs, connection arguments can be written easily in PIL by introducing a column associated with the chosen partition. This is also done in [[GWC19]](https://eprint.iacr.org/2019/953). Recall that column values are evaluations of a polynomial at $G = \langle g \rangle$ and $\texttt{N}$ is the length of the execution trace. Given a polynomial $\texttt{a}$ and a partition ${\large{\S}}$, suppose we want to write in PIL a constraint attesting to the $\textit{copy-satisfiability}$ of a certain ${\large{\S}}$. We first construct a permutation $\sigma : [n] \to [n]$ such that for each set $Si \in {\large{\S}}$, we have that $\sigma({\large{\S}})$ contains a cycle of all elements of Si. In the above example, we would have $\sigma = (5, 2, 1, 6, 3, 4)$. So then, we construct a polynomial Sa that encodes σ in the exponent of g. That is: $$ S{\texttt{a}}(g^i) = g^{\sigma(i)} $$ for $i \in [n]$. In the PIL context, the previous connection argument between a column $\texttt{a}$ and a column $\texttt{SA}$, encoding the values of $S{\texttt{a}}$, can be declared using the keyword $\texttt{connect}$ using the syntax: $\texttt{a}\ \texttt{connect}\ \{\texttt{SA}\}$. include "config.pil"; namespace Connection(%N); pol commit a; pol constant SA; {a} connect {SA}; A valid execution trace for this example was shown in Table 1 above. !!! info Remark The column $\texttt{SA}$ does not need to be declared as a constant polynomial. The connection argument still holds true even if it is declared as committed. Multiple copy satisfiability Connection arguments can be extended to several columns by encoding each column with a â€œpartâ€ of the permutation. Informally, the permutation is now able to span across the values of each of the involved polynomials in a way that the cycles formed in the permutation must contain the same value. Multi-column copy satisfiability Given vectors $a1, \dots , ak$ in $\mathbb{F}^n$ and a partition ${\large{\S}} = \{S1,...,St\}$ of $[kn]$, we say $a1,...,ak$ $\textit{copy-satisfy}\ {\large{\S}}$ if for each $Sm \in {\large{\S}}$, we have that $a{(l1},i) = a{(l2},j)$ whenever $i,j \in Sm$, with $i,j \in [n]$, $l1, l2 \in [k]$ and $m \in [t]$. For example, say that we have ${\large{\S}} = \{\{1\}, \{2,3,4,9\}, \{5\}, \{6\}, \{7,10\}, \{8,11\}, \{12\}\}$. Then, the below table depicts an execution trace for three columns $\texttt{a}, \texttt{b}, \texttt{c}$ that copy-satisfies ${\large{\S}}$. !An execution trace subject to a connection argument We reduce this problem to the one column case by thinking of the permutation σ as applied to the concatenation of column $\texttt{a}$, then $\texttt{b}$ and finally $\texttt{c}$. So, the permutation σ that makes $\texttt{a}$, $\texttt{b}$ and $\texttt{c}$ copy-satisfy ${\large{\S}}$ is $(1, 9, 2, 3, 5, 6, 10, 11, 4, 7, 8, 12)$. In this case we construct polynomials $S{\texttt{a}}$, $S{\texttt{b}}$ and Sc such that: $$ S{\texttt{a}}(g^i)\ = g^{\sigma(i)},\ S{\texttt{b}}(g^i)\ = k1 \cdot g^{\sigma(n+i)},\ S{\texttt{c}}(g^i)\ = k2 \cdot g^{\sigma(2n+i)} $$ where $k1, k2 \in \mathbb{F}$ are introduced here as a way of obtaining more elements (in a group G of size n) and enabling correct encoding of the $[3n] \to [3n]$ permutation σ. See [[GWC19]](https://eprint.iacr.org/2019/953) for more details on this encoding. The below table shows how to compute the polynomials $\texttt{SA}$, $\texttt{SB}$ and $\texttt{SC}$ encoding the permutation of the above example: !Multi-column connection argumentâ€™s valid execution trace The PIL code for this example is easily written as follows: include "config.pil"; namespace Connection(%N); pol commit a, b, c; pol constant SA, SB, SC; { a, b, c } connect { SA, SB, SC }; # cyclicity-in-pil.md: This document describes how to introduce cyclicity to execution traces in Polynomial Identity Language. In order to synchronize the execution trace of a given program with the subgroup G of the multiplicative group $\mathbb{F}^$, over which interpolation is performed, an extra constant polynomial (or precompiled column) is added to the trace. An explanation of what this group $G = \langle g \rangle$ is, and why it is naturally a cyclic group, was discussed in the Basic Concepts section of the zkProver. Non-cyclic SM example Consider a program with the following execution trace of length $\%\texttt{N} = 4$; $$ \begin{aligned}\begin{array}{|l|c|c|}\hline \texttt{row} &\ \mathtt{a} & \mathtt{b} \\ \hline \ \text{ 0} & \ \texttt{1} & \texttt{1} \\ \hline \ \text{ 1} & \ \texttt{0} & \texttt{2} \\ \hline \ \text{ 2} & \texttt{-1} & \texttt{2} \\ \hline \ \text{ 3} & \ \texttt{1} & \texttt{1} \\ \hline \end{array} \end{aligned} $$ Denote the cyclic group over which interpolation is carried out as $G = \{ g, g^2, g^3, g^4 = 1 \} \subset \mathbb{F}$. !!! info Concession to an abuse of notation We use the symbols $\texttt{a}$ and $\texttt{b}$, that denote columns of the execution trace, to also denote the corresponding polynomials resulting from interpolation. The columns $\texttt{a}$ and $\texttt{b}$ are best expressed as arrays, $$ \texttt{a} = [1,0,-1,1] \ \text{and}\ \texttt{b} = [1,2,2,1] $$ while the respective polynomials that result from interpolation, should rather be denoted differently, say with, $P(X)$ and $Q(X)$, such that for each row index i, $$ P(g^i) = \texttt{a}[i]\ \ \text{and}\ \ Q(g^i) = \texttt{b}[i] \tag{eqn} $$ But in order to keep the PIL code simple and easily relatable to the execution trace, we replace the P and Q with $\texttt{a}$ and $\texttt{b}$, respectively. The above eqn is therefore seen written as, $$ \texttt{a}(g^i) = \texttt{a}[i]\ \ \text{and}\ \ \texttt{b}(g^i) = \texttt{b}[i]. $$ For the sake of convenience, in this particular example, the row index starts at 0 just so it syncs with the normal array indexing. Constraints and cyclicity Observe that the column $\mathtt{a}$ only takes on the values; 0, 1 or $âˆ'1$; and these values satisfy the constraint, $$ (\mathtt{a} + 1)\cdot\mathtt{a}\cdot(\mathtt{a} - 1) = 0, $$ while the values of $\texttt{b}$ are such that $$ \texttt{b}' = \texttt{a}' + \texttt{b}. $$ This second constraint should be interpreted as, $$ \texttt{b}'(g^{i}) = \texttt{a}(g^i) + \texttt{b}(g^i). $$ However, this second constraint is satisfied for every row except for the last one. That is, $\texttt{b}'(g^{3}) \not= \texttt{a}(g^3) + \texttt{b}(g^3)$. Let us proof this inequality. But we first check one of the other cases. In particular, the case for $i = 2$. Recall that $\texttt{b}'(g^{i}) = \texttt{b}(g\cdot g^{i}) = \texttt{b}(g^{i+1})$ by definition. Then, $$ \text{LHS} = \texttt{b}'(g^{2}) = \texttt{b}(g\cdot g^{2}) = \texttt{b}(g^3) = b[3] = 1 \ \ \text{and} \\ \text{RHS} = \texttt{a}(g^{2}) + \texttt{b}(g^{2}) = a[2] + b[2] = -1 + 2 = 1 $$ This proves that the second constraints holds true for the case $i = 2$. Now for the case $i = 3$. Note that $g^{4} = 1 = g^0$, and again by definition, $\texttt{b}'(g^{3}) = \texttt{b}(g\cdot g^{3}) = \texttt{b}(g^{4}) = \texttt{b}(g^{0})$. So then, $$ \text{LHS} = \texttt{b}'(g^{3}) = \texttt{b}(g\cdot g^{3}) = \texttt{b}(g^{0}) = \texttt{b}[0] = 1\ \text{and} \\ \text{RHS} =\ \texttt{a}(g^3) + \texttt{b}(g^3) =\ \texttt{a}[3] + \texttt{b}[3] = 1 + 1 = 2. $$ This proves that second constraint is not satisfied for $i = 3$. And therefore the execution trace is not cyclic. Introducing cyclicity The execution trace can be made cyclic by introducing a selector polynomial, call it $\texttt{SEL}$, such that its column values are 1 in every row except the last, where it's 0, i.e., $\texttt{SEL}[i] = 1$ for all $i \in \{ 0, 1, 2 \}$, otherwise, $\texttt{SEL}[3] = 0$. The above execution trace is now modified to the following: $$ \begin{aligned} \begin{array}{|l|c|c|c|}\hline \texttt{row} & \ \mathtt{a} & \mathtt{b} & \texttt{SEL} \\ \hline \ \text{ 0} & \ \texttt{1} & \texttt{1} & \texttt{1}\\ \hline \ \text{ 1} & \ \texttt{0} & \texttt{2} & \texttt{1}\\ \hline \ \text{ 2} & \texttt{-1} & \texttt{2} & \texttt{1}\\ \hline \ \text{ 3} & \ \texttt{1} & \texttt{1} & \texttt{0}\\ \hline \end{array} \end{aligned} $$ Given these adjustments, we note that; - For all $i \in \{ 0, 1, 2, 3 \}$, the constraint $(\mathtt{a} + 1)\cdot\mathtt{a}\cdot(\mathtt{a} - 1) = 0$ still holds true. - For all $i \in \{ 0, 1, 2 \}$, the constraint $\texttt{b}' = \texttt{a} + \texttt{b}$ holds true as before. And, even if the constraint is adjusted to $\texttt{b}' = \texttt{SEL} \cdot (\texttt{a} + \texttt{b})$, it still holds true for all $i \in \{ 0, 1, 2 \}$. - For $i = 3$, we would like to have $\texttt{b}'(g^3) = \texttt{b}(g^0) = \texttt{b}[0] = 1$. Note that $\texttt{SEL}[3] = 0$, for this particular case. And hence, $$ \texttt{SEL} \cdot (\texttt{a} + \texttt{b}) = 0 \cdot (\texttt{a} + \texttt{b}) = 0. $$ All-in-all, the adjusted execution trace attains cyclicity if the second constraint is set to: $$ \texttt{b}' =\ \texttt{SEL} \cdot (\texttt{a} + \texttt{b})\ +\ (1 - \texttt{SEL}). $$ As seen earlier, for the case $i = 3$, $$ \text{LHS} = \texttt{b}'(g^{3}) = \texttt{b}(g\cdot g^{3}) = \texttt{b}(g^{0}) = \texttt{b}[0] = 1 $$ and now, $$ \text{RHS} =\ \texttt{SEL}(g^3)\cdot \big(\texttt{a}(g^3) + \texttt{b}(g^3)\big) +\ \big(1 - \texttt{SEL}(g^3)\big) \\ =\ \texttt{SEL}[3] \cdot \big(\texttt{a}[3] + \texttt{b}[3] \big) +\ (1 - \texttt{SEL}[3])\ \text{ } \\ =\ 0\cdot \big(1 + 1 \big) + \big(1 - 0 \big) =\ 0 + 1 = 1.\quad $$ The valid PIL code, with all the adjustments, is as depicted below: namespace CyclicExample(4); pol commit a, b; pol constant SEL; pol carry = (a+1)a; carry(a-1) = 0; b' = SEL(b+a) + (1-SEL); For implementation purposes, even as alluded to in the previous section, in order to prevent exposing distinguishing features, a configuration file is used to store the exact length of the program $\%\texttt{N} = 4$ so that only the symbol $\texttt{N}$ appears in the PIL code. # filling-polynomials.md: This document describes how to fill Polynomials in PIL using JavaScript and Pilcom. In this document, we are going to use Javascript and pilcom to generate a specific execution trace for a given PIL. To do so, we are going to use the execution trace of a program previously discussed in the Connection arguments section. We also use the pil-stark library, which is a utility that provides a framework for setup, generation and verification of proofs. It uses an FGL class which mimics a finite field, and it is required by some functions that provide the pilcom package. Execute code First of all, under the scope of an asynchronous function called execute, we parse the provided PIL code (which is, in our case, main.pil) into a Javascript object using the compile function of pilcom. In code, we obtain the following; js const { FGL } = require("pil-stark"); const { compile } = require("pilcom"); const path = require("path"); async function execute() { const pil = await compile(FGL, path.join(dirname, "main.pil")); } Pilcom package The pilcom package also provides two functions; newConstPolsArray and newCommitPolsArray. Both these functions use the pil object in order to create two crucial objects: 1. First is the constant polynomials object constPols, which is created by the newConstPolsArray function. 2. Second is the committed polynomials object cmPols, created by newCommitPolsArray. Below is an outline of the pilcom package. js const { newConstantPolsArray, newCommitPolsArray, compile } = require("pilcom"); async function execute() { // ... Previous Code const constPols = newConstantPolsArray(pil); const cmPols = newCommitPolsArray(pil); } Accessing execution trace The above-mentioned objects contain useful information about the PIL itself, such as the provided length of the program N, the total number of constant polynomials and the total number of committed polynomials. Accessing these objects allows us to fill the entire execution trace for that PIL. A specific position of the execution trace can be accessed by using the syntax: js pols.Namespace.Polynomial[i] Note that; - pols points to one of the above-mentioned objects; constPols and cmPols objects - Namespace is a specific namespace among the ones defined by the PIL files - Polynomial refers to one of the polynomials defined under the scope of the namespace - index i is an integer in the range $[0, N âˆ' 1]$, representing the row of the current polynomial Using these, the polynomials can now be filled. Main.pil code example In our example, we recall the main.pil seen in the Connection arguments section about 4-bit integers. Since we are only allowed to use 4-bit integers, inputs for the trace, which are also the ones introduced in the $\texttt{Main.a}$ polynomial, is a chain of integers n ascending cyclically from 0 to 15. We propose two functions here. - One for building the constant polynomials: js async function buildConstantPolynomials(constPols, polDeg) { for (let i=0; i < polDeg; i++) { constPols.Global.BITS4[i] = BigInt(i & 0b1111); constPols.Global.L1[i] = i === 0 ? 1n : 0n; constPols.Negation.RESET[i] = (i % 4) == 3 ? 1n : 0n; constPols.Negation.FACTOR[i] = BigInt(1 << (i % 4)); constPols.Negation.ISLAST[i] = i === polDeg-1 ? 1n : 0n; } } - And another function for building the committed polynomials: js async function buildcommittedPolynomials(cmPols, polDeg) { cmPols.Negation.a[-1] = 0n; cmPols.Negation.nega[-1] = 1n; for (let i=0; i < polDeg; i++) { let fourBitsInt = i % 16; cmPols.Main.a[i] = BigInt(fourBitsInt); cmPols.Main.nega[i] = BigInt(fourBitsInt ^ 0b1111); cmPols.Main.op[i] = FGL.mul(cmPols.Main.a[i], cmPols.Main.nega[i]); cmPols.Multiplier.freeIn1[i] = cmPols.Main.a[i]; cmPols.Multiplier.freeIn2[i] = cmPols.Main.nega[i]; cmPols.Multiplier.out[i] = cmPols.Main.op[i]; let associatedInt = Math.floor(i/4); let bit = (associatedInt >> (i%4) & 1) % 16; cmPols.Negation.bits[i] = BigInt(bit); cmPols.Negation.nbits[i] = BigInt(bit ^ 1); let factor = BigInt(1 << (i % 4)); let reset = (i % 4) == 0 ? 1n : 0n; cmPols.Negation.a[i] = factorcmPols.Negation.bits[i] + (1n-reset)cmPols.Negation.a[i-1]; cmPols.Negation.nega[i] = factorcmPols.Negation.nbits[i] + (1n-reset)cmPols.Negation.nega[i-1]; } } Once the constant and committed polynomials have been filled in, we can check whether these polynomials actually satisfy the constraints defined in the PIL file, by using a function called verifyPil. Below is the piece of code that constructs the polynomials and checks the constraints. If the verification procedure fails, we should not proceed to the proof generation because it leads to false proof. For the sake of brevity, we simply use the line // ... Previous Code to indicate where the PIL code being verified would be inserted. js const { newConstantPolsArray, newCommitPolsArray, compile, verifyPil } = require("pilcom"); async function execute() { // ... Previous Code const N = constPols.Global.BITS4.length; await buildConstantPolynomials(constPols, N); await buildcommittedPolynomials(cmPols, N); const res = await verifyPil(FGL, pil, cmPols , constPols); if (res.length != 0) { console.log("The execution trace do not satisfy PIL restrictions. Aborting..."); for (let i=0; i". 0 => A,B 2. Similarly, we can store the value of a register into other registers. A => B,C More generally, we can store the value of a function f of registers. f(A,B) => C,D 3. We can also store a global variable into some register. %GLOBALVAR => A,B 4. The result of executing an executor method can also be stored into one or more registers. The indication of such an execution is done with the dollar "$" sign, which should be treated as a free input. ${ExecutorMethod(params)} => A,B ; Notice that the method ExecutorMethod does not necessarily depend on the registers. ; A good example of such a method is SHA256. 5. If a method gets executed (with the dollar sign) by its own, its main purpose is generating log information. ${ExecutorMethod(params)} 6. Apart from executor methods, one can also use inline functions. These functions, which are also instantiated by the executor, are simply "short" and non-reused executor methods. ${A >> 2} => B ${A & 0x03} => C Introducing opcodes Until this point, every instruction consisted of a direct interaction with the registers. Now, we move one step forward and we create interaction with other parts of the ROM, thanks to the introduction of zkEVM Opcodes. To assign the output of a zkEVM Opcode into some register, we use the following syntax: $ => A,B :OPCODE(param) A clear example of one such situation is while using the memory load opcode: $ => A,B :MLOAD(param) When a registers appear at the side of an Opcode, it is typically used to indicate that the value of the register A is the input of the memory store opcode: A :MSTORE(param) Similarly, we can assign a free input into a register and later on execute several zkEVM Opcodes using the following syntax: ${ExecutorMethod(params)} => A :OPCODE1 :OPCODE2 :OPCODE3 ... When an executor method with a register to store its result gets combined with a jump opcode, it is typically used to handle some unexpected situation, e.g. running out of gas: ${ExecutorMethod(params)} => A :JMP(param) It is also common to encounter negative jumps to check appropriate situations, in which carry on forthcoming operations: SP - 2 :JMPN(stackUnderflow) Code injection Inline javascript-based instruction can be injected in plain by using the double dollar "$" symbol. $${CODE} The main difference between the single dollar sign and the double dollar sign is that while the methods inside the single dollar sign come from the Executor, the double dollar ones do not. It's a plain javascript code that is executed by the ROM. Asserts Asserts work by comparing what is being asserted with the value on register A. For instance, the following instructions compares the value inside register B with the value inside register A: B :ASSERT # examples.md: EVM ADD Let's take the EVM ADD opcode as *our first starting example: opADD: SP - 2 :JMPN(stackUnderflow) SP - 1 => SP $ => A :MLOAD(SP--) $ => C :MLOAD(SP) ; Add operation with Arith A :MSTORE(arithA) C :MSTORE(arithB) :CALL(addARITH) $ => E :MLOAD(arithRes1) E :MSTORE(SP++) 1024 - SP :JMPN(stackOverflow) GAS-3 => GAS :JMPN(outOfGas) :JMP(readCode) Here is a detailed explanation of how the ADD opcode gets interpreted. Recall that at the beginning, the stack pointer is pointing to the next "empty" address in the stack: 1. First, we check if the stack is filled "properly" in order to carry on the ADD operation. This means that, as the ADD opcode needs two elements to operate, it is checked that these two elements are actually in the stack: SP - 2 :JMPN(stackUnderflow) If less than two elements are present, then the stackUnderflow function gets executed. 2. Next, we move the stack pointer to the first operand, load its value and place the result in the A register. Similarly, we move the stack pointer to the next operated, load its value and place the result in the C register. SP - 1 => SP $ => A :MLOAD(SP--) $ => C :MLOAD(SP) 3. Now its when the operation takes place. We perform the addition operation by storing the value of the registers A and C into the variables arithA and arithB and then we call the subroutine addARITH that is the one in charge of actually performing the addition. A :MSTORE(arithA) C :MSTORE(arithB) :CALL(addARITH) $ => E :MLOAD(arithRes1) E :MSTORE(SP++) Finally, the result of the addition gets placed into the register E and the corresponding value gets placed into the stack pointer location; moving it forward afterwise. 4. A bunch of checks are performed. It is first checked that after the operation, the stack is not full and then that we do not run out of gas. 1024 - SP :JMPN(stackOverflow) GAS-3 => GAS :JMPN(outOfGas) :JMP(readCode) Last but not the least, there is an instruction indicating to move forward to the next instruction. # index.md: Ethereum is a state machine that transitions from an old state to a new state by reading a series of transactions. It is a natural choice, in order to interpret the set of EVM opcodes, to design another state machine as for the interpreter. One should think of it as building a state machine inside another state machine, or more concretely, building an Ethereum inside the Ethereum itself. The distinction here is that the former contains a virtual machine, the zkEVM, that is zero-knowledge friendly. zkEVM as a microprocessor Following the previous discussion, it is good to see the outer state machine as a microprocessor. What we have done is creating a microprocessor, composed by a series of assembly instructions and its associate program (i.e., the ROM) running on top of it, that interprets the EVM opcodes. Below provided is the block diagram of a basic uniprocessor-CPU computer. Black lines indicate data flow, whereas red lines indicate control flow; arrows indicate flow directions. `![](../../../img/zkEVM/CPU.png)` As in input, the microprocessor will take the transactions that we want to process and the old state. After fetching the input, the ROM is used to interpret the transactions and generate a new state (the output) from them. Check out the diagram below for a better visualization. `![](../../../img/zkEVM/machine-cycle.png)` The role of zkASM The zero-knowledge Assembly (zkASM) is the language used to describe, in a more abstract way, the ROM of our processor. Specifically, this ROM will tell the Executor how to interpret the distinct types of transactions that it could possibly receive as an input. From this point, the Executor will be capable of generating a set of polynomials that will describe the state transition and will be later on used by the STARK generator to generate a proof of correctness of this state transition. `![](../../../img/zkEVM/big-picture.png)`*