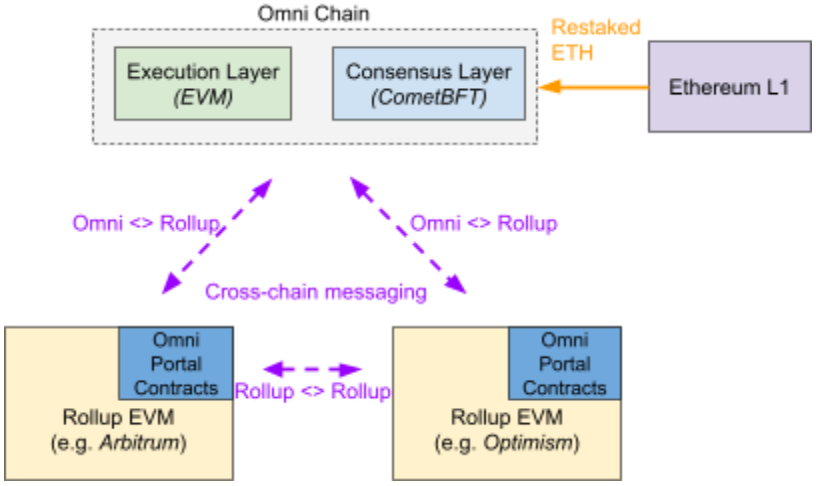
Omni Protocol Overview for Security Reviewers

The purpose of this document is to outline the design of the first version of the Omni Protocol. It describes the main components, their interactions, and the set of features they provide.

Please note that some of these comments might be slightly out of date by the time you're reading this. In particular, the best source of truth for strictly defined types is the code base itself.

Overview

The Omni protocol v0 provides a set of smart contracts deployed on the Omni Chain EVM and on EVM-compatible Ethereum rollups that expose an interface to perform cross-chain smart contract calls between these EVMs. This cross-chain messaging is secured by the Omni Chain's consensus layer dPoS validator set that is backed by the Omni Chain's native $OMNI token (and in a future release, by re-staked ETH from Ethereum L1 via Eigenlayer)



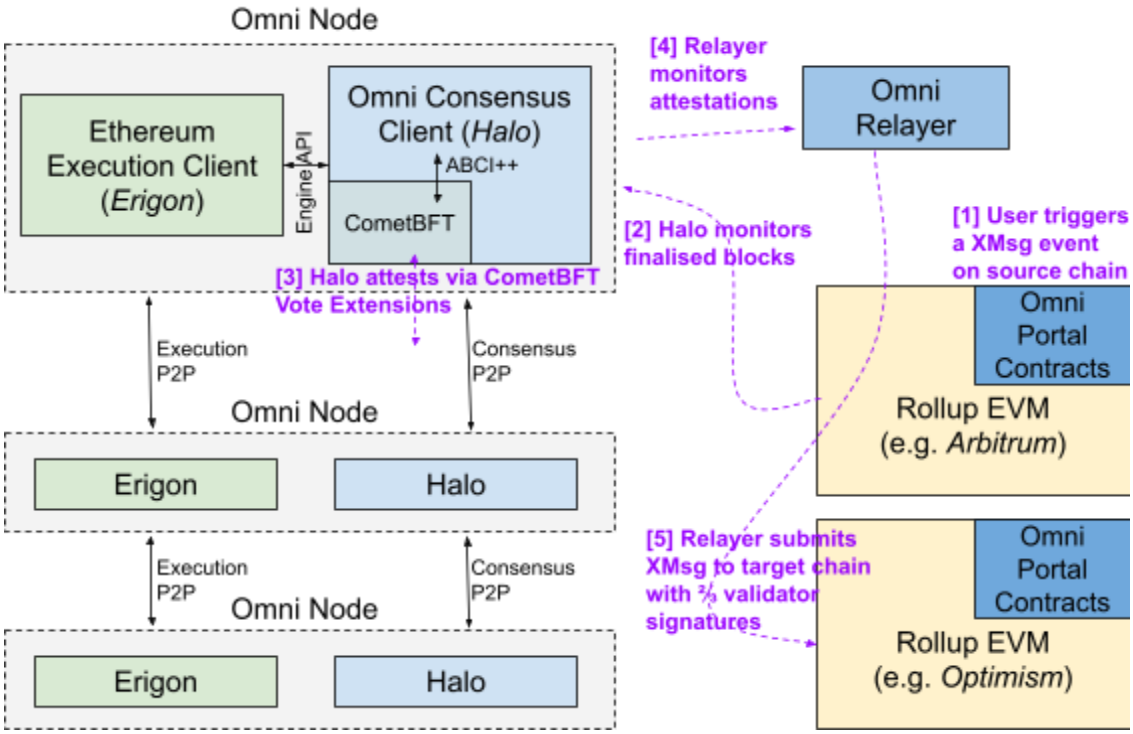
Components

- **Rollup EVM**:
  - Rollup EVMs represent public Ethereum L2 EVMs like Arbitrum, Optimism, Base.
  - Omni protocol facilitates cross chain messaging between rollup EVMs (+ Ethereum and Omni EVM).
  - The rollups expose the standard EVM JSON-RPC APIs that different components in the omni protocol can query to discover the latest remote state and to submit transactions to modify that state.
- **Omni Chain:**

    - Omni Chain is a L1 blockchain consisting of two internal chains, a consensus layer and an execution layer, similar to post-merge Ethereum.
    - The execution layer is implemented by a standard Ethereum execution client providing the Omni EVM (geth).
    - The consensus layer is implemented by the Omni Consensus client, halo, which is a cosmos-sdk chain used to provide security for cross-chain messaging and for the Omni execution layer.
- **Omni EVM**:
    - The Omni EVM is an Ethereum compatible EVM implemented by the Omni chain's execution layer.
    - It is implemented by any standard Ethereum execution client like geth or Erigon (vanilla).
    - Omni protocol facilitates cross chain messaging between rollup EVMs and the Omni EVM.
    - It will be used for native $OMNI token issuance and staking.
    - It is publicly accessible like any permissionless EVM.
    - It could be used as the central coordination layer from cross-chain dapps that prefer a hub-and-spoke model.
- **Omni Consensus Client (Halo)**:
    - The first implementation of the omni consensus layer is called Halo.
    - It uses CometBFT with dPoS which is secured by native $OMNI with delegation by restaked $ETH from Ethereum L1.
    - It is a cosmos-sdk chain, with ~8 custom modules to enable various building blocks for cross-chain messaging and the Omni EVM.
    - It implements the server side of the ABCI++ interface.
    - It drives the Omni Execution Layer via the Engine API.
    - Validators attest to source chain blocks containing cross chain messages using CometBFT Vote Extensions.
- **Portal Contracts**
    - A set of smart contracts that implements the on-chain logic of the Omni protocol.
    - Deployed to all supported Rollup EVMs as well as the Omni EVM and Ethereum L1.
    - Provides the main interface to "call a cross-chain smart contract" which results in a cross-chain message being "emitted" via a `XMsg` event log.
    - Provides v1 "pay at source" fee mechanism using the source chain's native token.
    - Tracks the omni consensus validator set, used to verify submitted cross chain message attestations.
- **Relayer**
    - Relayer is a permissionless actor that submits cross chain messages to destination chains.
    - Monitors the Omni Consensus Layer until ⅔ (>66%) of the validator set attested to the "next" xblock on each source chain.

- Submits the applicable cross-chain messages to each destination chain providing the quorum validator signatures and a multi-merkle-proof.
- Will eventually be incentivized.

Architecture Diagram



Architecture diagram with cross chain message flow

Cross Chain Messaging

The cross chain message flow can be decomposed into the following steps:

1. **User triggers `XMsg` event on a source chain**
    - An `xcall` function is called on one of the Omni Portal Contracts which logs the following `XMsg` event.
    - Note that this assumes that fees have already been paid at this point.
    - `XMsg` events are included in source chain blocks.
    - `XMsgs` are associated with an `XStream`. An `XStream` is a logical connection between a source and destination chain. It contains `XMsgs`, each with a monotonically incrementing `XStreamOffset` (the offset is like a EOA nonce, it is incremented for each subsequent message sent from a source chain to a destination chain). `XMsgs` are therefore uniquely identified and strictly ordered by their associated `XStream` and `Offset`. An `XStream` is uniquely identified by a `SourceChainID`, `DestChainID`, and `ConfLevel`.

```
None
event XMsg(
       uint64   destChainId  // Target chain ID as per https://chainlist.org/
       uint64   shardId      // Shard ID of the XStream (first byte is the
confirmation level)
       uint64   offset       // Monotonically incremented offset of XMsg in the
XStream
       address  sender       // Sender on source chain, from msg.sender
       address  to           // Target/To address to "call" on destination chain
       bytes    data         // Data to provide to "call" on destination chain
       uint64   gasLimit     // Gas limit to use for "call" on destination chain
       uint256  fees         // Fees paid for the xcall
)
```

- `XStreamOffset` allows exactly-once delivery guarantees with strict ordering per source-destination chain pair.

2. **Halo monitors finalized and latest blocks**
   - Each Omni consensus layer validator monitors every finalized and latest block for all source chains.
   - Note that validators need to wait for block finalization, or some other agreed-upon threshold, to ensure consistent and secure cross-chain messaging.
   - The `finalized` and `latest` streams are both offered (can be selected by developers per XMsg). The `finalized` stream provides exactly once delivery guarantees, while the `latest` stream provides no delivery guarantees.
   - Each source chain block is deterministically converted into the following `XBlock` structure. It is a deterministic one-to-one mapping.

```
None
// XBlock represents the cross-chain properties of a source chain finalised
block.
type XBlock (
       XBlockHeader
       Msgs        []Msg       // All xmessages sent/emitted in the block
       Receipts    []Receipt   // Receipts of all submitted xmessages in the
block
       ParentHash  common.Hash // ParentHash is the hash of the parent block.
       Timestamp   time.Time   // Timestamp of the source chain block
}

type XBlockHeader (
       ChainID     uint64      // Source chain ID as per https://chainlist.org
       BlockHeight uint64      // Height of the source-chain block
```

```
        BlockHash   common.Hash // Hash of the source-chain block
)
```

- `XBlock` structure provide the following properties to the Omni Protocol:
  i. Succinctly verifiable merkle-multi-proofs for sub-ranges of `XMsgs` per source-target pair allowing relayers to manage submission costs at single `XMsg` granularity.
  ii. Omni Consensus attestations are not required for source chain blocks without any cross chain messages (aka empty `XBlocks`).
  iii. Relayer submissions are not required on destination chains for batches without cross chain messages (aka empty `XBlocks`).
- The logic to create a `XBlock` is deterministic for any finalized source chain block height.
- Note the `XReceipts` are introduced and discussed later in the flow.

3. **Halo attests via CometBFT vote extensions**
   - All validators in the CometBFT validator set should vote for all `XBlocks` (in addition to their normal validator duties).
   - Quorum votes constitute an approved attestation.
   - A vote is defined by the following `Vote` type.

```
None
type Vote (
        AttestHeader attest_header   // uniquely identifies an attestation
        BlockHeader  block_header    // BlockHeader identifies the XBlock
        bytes        msg_root        // Merkle root of all xmsgs in the XBlock
        SigTuple     signature       // Validator signatures and public keys
)

type AttestHeader (
        uint64 consensus_chain_id // Omni c-chain ID this attestation/vote
        belongs to
        uint64 source_chain_id    // Source Chain ID as per https://chainlist.org
        uint32 conf_level         // Confirmation level (aka version) of the
        xblock.
        uint64 attest_offset      // Monotonically increasing offset
)

type BlockHeader (
        uint64 chain_id      // Source chain ID as per https://chainlist.org
```

```
       uint64 block_height  // Height of the source-chain block
       bytes  block_hash    // Hash of the source-chain block
)

type SigTuple (
       bytes validator_address // Validator ethereum address; 20 bytes.
       bytes signature         // Validator signature over AttestationRoot;
)
```

- Validators return an array of `Votes` during the ABCI++ `ExtendVote` method.
- Validators should reject vote extensions that contain invalid votes via `VerifyVoteExtension`.
- Proposers include the `Votes` from the previous block into an array of `AggregateVotes` that are included in the `CPayload` type (see below) during `PrepareProposal`. `AggregateVotes` simply removes block header field duplication and therefore decreases `CPayload` size.
- Omni Consensus clients process the consensus blocks and maintain the status of each `Attestation` by merging any new votes into it and updating the `Approved` status once quorum votes are included for the `ValidatorSetID`. This should be made available for querying to Relayers.

None

```
// Attestation contains quorum votes for a cross-chain block of a specific
validator set.
type Attestation (
       AttestHeader      attest_header    // uniquely identifies an attestation
       BlockHeader       block_header     // identifies the XBlock
       bytes             msg_root         // Merkle root of all xmsgs in the
XBlock
       SigTuple[]        signatures       // Validator signatures and public
keys
       uint64            validator_set_id // Validator set that approved this
attestation.
)


// AggVote aggregates multiple votes of an XBlock.
type AggVote (
       AttestHeader      attest_header   // uniquely identifies an attestation
       BlockHeader       block_header    // BlockHeader identifies the XBlock
```

```
      bytes             msg_root        // Merkle root of all xmsgs in the
XBlock
      SigTuple[]  signatures      // Validator signatures and public keys
)
```

- Validators in the current validator set must vote for all subsequent (after the “last approved”) `Attestations`.
- When the validator set changes, all “pending” `Attestations` need to be updated by:
  i. Updating the associated validator set ID to the current.
  ii. Deleting all attestations by validators not in the current set.
  iii. Updating the weights of each remaining attestation according to the new validator set.
- Validators that already voted for any pending `Attestation` during the previous validator set, do not need to re-attest. Only the new set validators must attest (ie. to all `XBlocks` after the latest approved).
- Only the “latest approved” `Attestation` for each source chain needs to be maintained in the consensus chain state; earlier `Attestations` can be trimmed from the state. (Relayers should just be able to query the old state if lagging).

4. **Relayer monitors attestations**
   - Similar to validators, relayers should maintain a `XBlock` cache. I.e., track all source chain blocks, convert them to `XBlocks,` cache them, and make them available for internal indexed querying.
   - Relayers should monitor the Omni Consensus Chain state for “approved” `Attestations`.
5. **Relayer submits `XMsgs` to destination chain with ⅔ validator signatures**
   - Relayers should then be able to trigger this step as soon as the “next” `Attestation` for any source chain is approved (has quorum signatures).
   - For each destination chain, the relayer has to decide how many `XMsgs` to submit, which defines the “cost” of transactions being submitted to the destination chain. This is primarily defined by the data size and gas limit of the messages and the portal contract verification and processing overhead.
   - A merkle-multi-proof is generated for the set of identified `XMsgs` that match the quorum `XBlock` attestations root.
   - The relayer submits a EVM transaction to the destination chain, ensuring it gets included on-chain as soon as possible.
   - The transaction contains the following data:

```
None
type Submission (
       bytes32      attestationRoot  // Merkle root of xchain block
       (XBlockRoot),
attested to and signed by validators
       uint64       validatorSetId   // identifier of the validator set that
       attested to
this root
       BlockHeader  blockHeader      // Block header, identifies xchain block
       Msg[]        msgs             // Messages to execute
       bytes32[]    proof            // Multi proof of block header and
       messages, proven
against attestationRoot
       bool[]       proofFlags      // Multi proof flags
       SigTuple[]   signatures       Array of validator signatures of the
attestationRoot, and their public keys
)
```

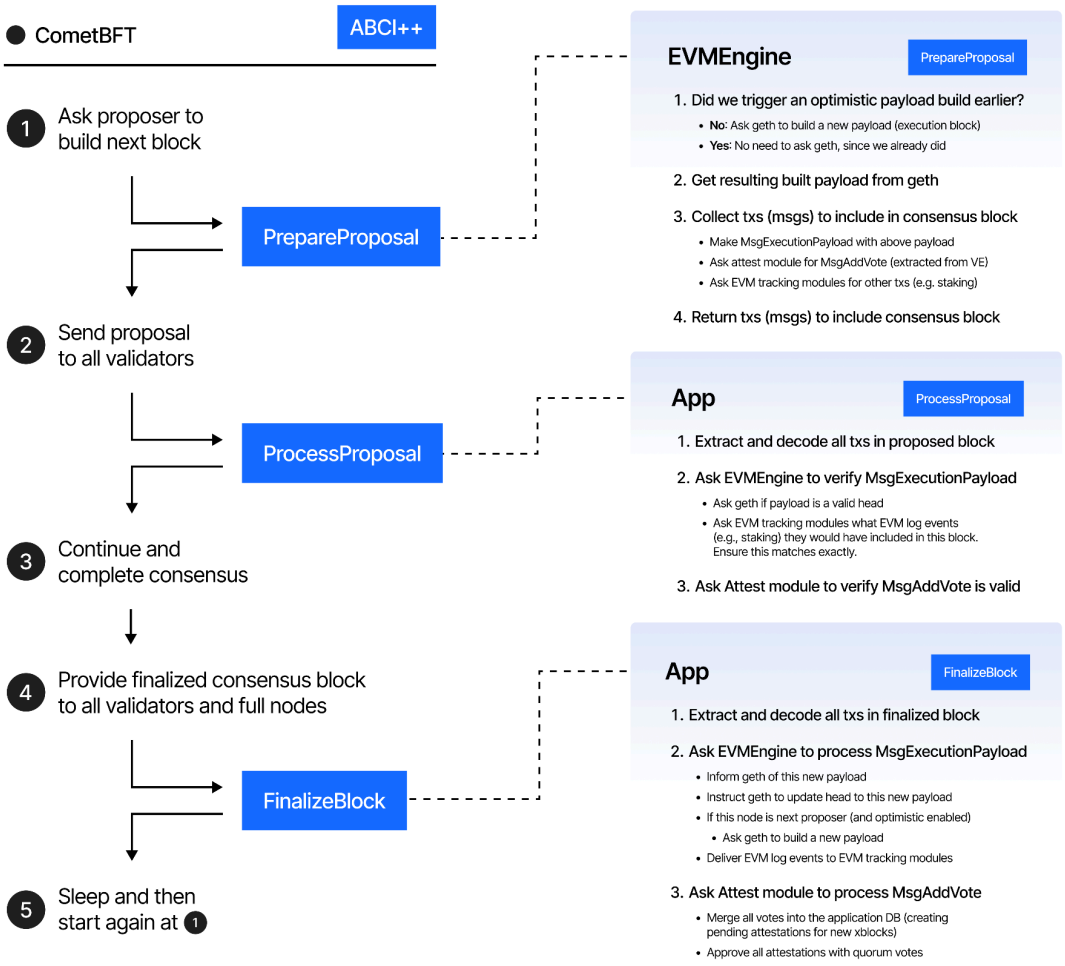
6. **Portal contract triggers `XReceipt` event on destination chain**
    - After validating and processing the submitted `XMsg`, the portal contract logs a `XReceipt` event.
    - This marks the `XMsg` as “successful” or “reverted”. `XMsgs` can revert if the gas limit was exceeded or if target address smart contract logic reverted for other reasons.
    - `XReceipts` are included in `XBlocks` (same as `XMsgs`). This is mostly as a convenience for cross chain explorers and end users. It isn’t used by the protocol itself.

```
None
type XReceipt (
       uint    SourceChainID         // The cross-chain message's source chain
       uint    XStreamOffset         // Offset of XMsg in the XStream
       uint    GasUsed               // Gas used during message "call"
       uint    Result                // 0 for success, 1 for revert
       address RelayerAddress        // Address of relayer that submitted the
message
)
```

Octane EVM

- Octane is a cosmos-sdk module that implements the consensus side of the EngineAPI

- It communicates with Geth or any other EVM execution client
- It processes blocks like Ethereum does, see [Engine API: A Visual Guide]. The main difference is that Halo uses CometBFT via the Cosmos SDK, while Ethereum uses Gasper (Casper FFG + LMD GHOST).



- The Omni EVM can be used by developers, but is also used for various purposes with in the Omni system
- **Predeploys**
    - The halo consensus chain is configured to watch several predeploy contracts on the Omni EVM – in particular Staking.sol, Slashing.sol, and Upgrades.sol
- **Staking**
    - The staking predeploy contract proxies the cosmos-sdk staking module
    - Validators can register, receive delegations, etc
    - Halo is configured to watch it via halo/evmstaking
- **Slashing**
    - The slashing predeploy contract proxies the cosmos-sdk slashing module
    - It allows validators to unjail themselves

    - Halo is configured to watch it via halo/evmslashing
- **Upgrades**
    - The Upgrades predeploy proxies cosmos-sdk upgrades
    - Halo is configured to watch it via halo/evmupgrade

FAQ

1. Why use CometBFT for consensus?
    - CometBFT (aka Tendermint) is a high-quality battle-tested general purpose blockchain consensus engine used in many production blockchain applications securing billions of dollars.
    - CometBFT is designed to work with delegated proof of stake, which fits our dual staking model of native $OMNI and re-staked L1 $ETH.
    - CometBFT has instant finality.
2. Why use an Ethereum execution client instead of ethermint as the EVM?
    - Post-merge ethereum decoupled the execution layer from the consensus layer introducing a more modular approach to building blockchains.
    - This modular approach allows the EVM to scale (somewhat) independently from consensus, by simply adopting the latest performant execution client like Erigon or reth.
    - Staying up to date with the latest upgrades in Ethereum is also much simpler, especially given that Ethermint has been abandoned for a long time.
3. Why implement the Omni EVM at all?
    - At time of writing, combining the Ethereum execution layer with CometBFT consensus would be a novel innovation that would enable new use-cases and provide value to the Ethereum community at large.
    - The Omni consensus layer needs smart contracts to manage native staking and delegated re-staking from ETH L1. The Omni EVM is a natural fit as fees would be much lower and syncing with the consensus layer is already built-in.
    - Providing an EVM purposely built for cross-chain dapps that has both low fees and short block times allows for a simple adoption path and hub-and-spoke mental model to onboard projects into Omni Protocol.
4. Instead of CometBFT Vote Extensions, why not follow Ethereum's Consensus Layer P2P subnet approach to collect and aggregate XBlock attestations?
    - Yes, the P2P-subnet approach could also work and would probably scale better than vote extensions.
    - Vote extensions are however easier to implement and should be sufficient for v1.
    - Further testing and analysis should be done to identify whether Vote Extensions should be refactored to P2P-subnets.
5. Why include `ValidatorSetID` in `Attestations`?
    - An `XBlock` should only be "approved" by a single set of validators.
    - Subsequent `XBlocks` should only be approved by the same or subsequent validator sets.

    - When portals verify a `Submission`, it needs to know the validator set to compare it to.
    - Portals only need to retain the validator sets for the latest `Submission` for each source chain. Older validator sets can be trimmed.
6. What do the `X*`, `C*` and `E*` type prefixes mean?
    - These prefixes indicate the “bounded context” which “own” the types.
    - `X*` indicates cross-chain layer types
    - `C*` indicates omni consensus layer types.
    - `E*` indicates omni execution layer types
7. How does the Portal Contract validate a `Submission`?
    - Portal Contracts keep a “cursor” for each source chain that:
        i. Tracks the latest valid `Submission`’s `XBlockHash` that contained valid `XMsgs` to the local destination chain.
        ii. The `Total` messages in that batch.
        iii. The `Index` of the last message that was submitted.
        iv. And implicitly, whether the latest `XBlock` is *partially* or *completely* submitted.
    - Validate the `Attestation` data:
        i. Ensure the `SourceChainID` is known?
        ii. Ensure the `ValidatorSetID` is known and the validator set is available.
        iii. If the cursor is *partial*, ensure the `XBlockHash` matches that of the cursor.
    - Validate the `XMsg` data:
        i. Ensure the `DestChainID` matches the local chain ID.
        ii. If the cursor is *complete*, ensure the `XStreamOffset` is the next expected value.
    - Verify the `Attestation` signatures:
        i. Verify all validator signatures over the root `XBlockHash`
        ii. Ensure that quorum is reached; more than 66% validators in the set signed.
    - Verify a merke-multi-proof against the `XBlockHash` that proves the following fields of the `XBlock` :
        i. All fields used in above validator.
        ii. All included `XMsgs` hashes.
8. Are `XBlocks` stored in the Omni execution or consensus layer? If so, which component is responsible for creating them and for setting `XReceipts` and `XMsgs` in them?
    - `XBlocks` are not stored anywhere. They are “deterministically calculated” from a source blockchain. So in effect, the source blockchain stores them.

    - Any component that depends on `XBlocks`, calculates it themselves from a source chain.
    - `XBlocks = f(chainA)` where `f(x)` is a deterministic “pure” function that takes a finalized blockchain as input and produces `XBlocks` as output.
    - In practice, source blocks can be streamed and transformed using a simple translation function backed by an in-memory cache.

Audit Notes

Areas of Interest

For Golang / Consensus Researchers

- Cosmos SDK Wiring: integrating cosmos sdk modules into halo/app, and ensuring configuration is done correctly for critical modules
- Valsync and light clients
    - The valsync module tracks validator set changes and propagates validator set updates to OmniPortal contracts on all supported chains in Omni
    - Each OmniPortal effectively runs a light client – it tracks the current (and last n) validator sets, and allows the current validator set to add new ones
    - It is critical that these validator set updates are propagated correctly
    - How does it do this? Well, the Portals already have a logical flow for confirming that a validator set has attested to some function call (XMsg). So the Omni Consensus chain uses the same logical flow – it packages the validator set update as an XMsg and sends it to each portal, which confirms that the validator set signed the XMsg containing the validator set update.
- Octane EVM Engine
    - Octane is the cosmos sdk module that runs the consensus side of Ethereum’s EngineAPI
    - It is responsible for communicating with the execution client and building EVM payloads, see above
- Attest module
    - This module watches omni portals for XMsgs (using lib/xchain), builds XBlocks, attests to them. It is the core logical component for cross-chain messaging.

For Solidity Researchers

- OMNI bridge
    - The OMNI bridge (under contracts/core/src/token) has 2 components – a contract on Ethereum and a contract on Omni
    - Each contract holds significant funds – the OMNI ERC20 on Ethereum, and the native token on Omni.
- xsubmit function

    - This function acts as the entrypoint for cross-chain calls. It must validate all XBlocks / XMsgs.
    - Correct validation of XBlocks and XMsgs is critical since if an invalid XBlock or XMsg can be submitted, the protocol is compromised.
- sysxcalls (system xcall)
    - The Omni Consensus Chain produces xmsgs relayed to all portals, executed at each portal. We call these “sysxcalls”. These currently include `setNetwork` and `addValidatorSet`
    - It’s critical that this sysxcall mechanism cannot be hijacked.

For Both

- XBlock data structure, merkle root, and merkle multi-proofs
    - Can an invalid XMsg be delivered?
- Confirmation strategies
    - Omni's xchain message protocol currently offers 2 confirmation strategies. Developers can specify their confirmation strategy with each xcall.
    - **Finalized** xmsgs are attested to and delivered only after the rollup's transaction data containing this xmsg finalizes on Ethereum Layer 1. This requires 2 beacon chain epochs, which typically takes about 12 minutes. However, this strategy offers strong delivery guarantees – a delivered message can only be "reorg'd out" if Ethereum itself reorgs, which is highly unlikely and requires 2/3 of Ethereum's validators to be slashed.
    - **Latest** xmsgs are attested to and delivered as soon as the transaction with the xmsg is included by the L2 sequencer in a block. This provides a much lower latency for message delivery – roughly 5-10s. However, it does come with an associated risk: the xmsg has a higher risk of being reorg'd out if the L2 sequencer misbehaves or fails. This may result in unintended consequences, and you should decide how much you're willing to trust L2 sequencers.

Known Issues

- The validator set is whitelisted in the V1 release. Validator actions are limited – there are no withdrawals, staking rewards, or delegations. It is assumed that there is always ⅔ quorum of honest validators.
- The validator allowlist is not planned to be removed in the current release, so no issues related to removing the allowlist will be considered.
- Blobs vs Calldata
    - FeeOracleV1/V2 is currently out of scope, as it does not currently take into account rollups that use blobs or non-EVM DA services.
- RANDAO opcode
    - Random attribute of the EVM payload is a predictable hash of the latest block.

    - This should be an actual random value. This will be implemented in a future release.
- Gas price oracles
    - The gas prices stored in `FeeOracleV1` are lagging and also might not accurately represent the gas prices at execution time.
    - Some xcalls may be "underpaid", though it's also true that some will be "overpaid", at an roughly equal rate.
- Stale streams
    - Context
        - Portals require that an XSubmission includes a validator set within the last 10 validator set IDs.
        - Validator set ID changes each time there is a new validator, a validator leaves, or an (un)delegation.
        - Note that this will happen infrequently in v1, since the validator set is whitelisted, and delegations are not yet enabled.
        - The relayer is an off chain component with a “1 of n” security model.
    - Risk:
        - If an XBlock B was signed by validator set V, AND
        - There were >10 validator set changes such that the current set if V+10 or greater, AND
        - The relayer failed to submit XBlock B to its destination by the time V+10 is active on the destination portal, THEN
        - the XStream will stall – because the submission will be using validator set V, and that is not within the last 10 validator sets
- Fee refunds
    - Each xcall checks that the user pays enough fees based on the destChainId, the data used in the xcall, and the gasLimit. While a user may accidentally or intentionally pay more than this required amount in the true execution on the destination, any excess payment will not be refunded.
    - Fee refunds are desirable, but will are out of scope for v1.0
- RPC Endpoints
    - Each validator runs full nodes for each integrated chain. Each validator trusts their RPC endpoints to return valid data.
- Staking and Unstaking
    - Staking and unstaking EVM events are currently batched roughly every 12 hours and processed by the consensus chain at that time.
    - Stakers and unstakers must provide 1 / 0.1 OMNI as a sybil prevention mechanism
    - Given it’s early in the network, we’ve decided this is sufficient in the short term, but this mechanism may be updated in the future to be more robust.
- Retry Mechanism
    - A “retry mechanism” is a way to retry a cross-chain message if it fails.

    - An in-protocol retry mechanism for failed cross-chain messages is out of scope for V1.
    - This can be built out-of-protocol in the short term. But likely will be added to the protocol medium-term.
- Overfilling EVM Blocks
    - In CometBFT, the `PrepareProposal()` function sets a limit on the maximum number of bytes that are allowed to fit in a proposed block. However, Halo is not allowed to remove any transactions from the proposed block if this limit is exceeded. If a Halo block were to exceed this limit, the chain would simply halt.
    - In practice, this can never happen. The maximum amount of gas in an EVM block is 30,000,000. The largest block possible when constrained by gas is a block filled with 0's. This results in a maximum block of 7,500,000 bytes, or 15,000,000 bytes when hex encoded in the EngineAPI.
    - Since `cmttypes.MaxBlockSizeBytes*9/10` evaluates to roughly 94 MB, it is impossible for a 15 MB maximum block size to ever exceed this.
- Bridge pausing
    - When a user calls bridge() to bridge OMNI tokens, an XMsg is sent to the destination chain to call the destination chain bridge's withdraw() function. However, if the withdraw function is paused on the destination chain after the XMsg is emitted, the XMsg will fail to be executed on the destination.
    - Deposits will always be paused before withdrawals (withdrawals will only be paused if users are able to withdraw without correct validation)
- XMsg Ordering
    - We rely on xmsgs being ordered by log index (ascending) to build the xblock merkle tree. This implicitly orders xmsgs by offset per shard. Order by log index is currently not enforced when constructing the xblock merkle tree. We have an open PR to enforce that ordering.
    - Note that portal xsubmission test utilities sort xmsgs per xblock by dest chain / offset.
- Syscall authorization
    - Syscalls are xmsgs to the `VirtualPortalAddress`. Currently, the only allowed system xcalls come from consensus chain xmsgs broadcasted to each portal. Verification in the portal contracts does not enforce xmsgs to the `VirtualPortalAddress` are broadcast from the consensus chain. This allows for more flexible syscalls, but as we do not yet need them, we plan to enforce more strict validation for syscalls.
- Malicious validators can send arbitrary geth messages such as for the fields BlobGasUsed, ExcessBlobGas, and others and stall the block confirmation process. This is due to the "shape of the payload" not matching "Deneb Payload". This is possible with current JSON encoding. But once we switch to protos, this won't be possible anymore
- Halo currently sets Block.MaxBytes==-1 which allows blocks up to 90% of cmttypes.MaxBlockSizeBytes which is 90MB.
    - This allows the following: Malicious proposer can propose very large invalid blocks, which use a lot of network/cpu/memory/disk resources which can slow down the chain since validating and rejecting these take a lot of resources.
- Prior Audits

    - Cantina review
    - Sigma Prime Part 1
    - Sigma Prime Part 2
    - Spearbit Solidity Audit
    - Spearbit Go Audit
    - Zellic Audit

Additional Resources

- Codebase Walkthrough
- Public Documentation
- Test Harnesses for smart contracts