Unicity EVM

Risto Laanoja / Jan 9 2025

Goal

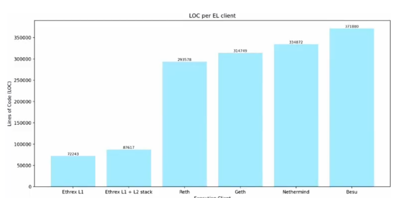
- EVM is the de-facto standard of DeFI, lots of copy-paste functionality, including external exchange integrations
- ERC-xxx standardized interfaces for assets
- Shared state use-cases, pooled assets (e.g. exchange liquidity pool, lottery)
- Parallel composability is simpler than sequential
- Programmable rules: governance, tokenomics (nothing interesting can be done on PoW chain)
- Familiar dev ux. Accounts for n00bs
- Transparency! for certain tokens/assets/...

Choices made

- rust because of ZK tooling
- zkVM because zkEVM-s are pain to extend with precompiles
- RISC-V because it is general (vs WASM/LISP/Cairo)
- SP1 because of 2nd mover benefits (the other production grade zkVM being RISC Zero)
- ethrex because of 80k LOC (vs crates from reth + revm)

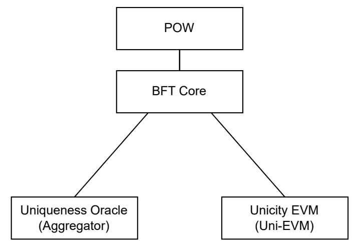
ethrex

- https://ethrex.xyz/ ; https://github.com/lambdaclass/ethrex
- L1 and "EVM Equivalent" L2 capability, pluggable proving backends (multiple ZK VMs, TEE (intel TDX))
- own evm - "levm"



Integrating with BFT Core

(infra-level integration)



- BFT Core works as a perfect L1 - it validates state transfers of underlying partitions (like L2s)
- It can support many partitions, like uniqueness oracles for different service classes, etc
- But it does not have a blockchain, just cumulative state, thus no L1 data availability.
- It returns Unicity Certificates for valid requests 'extending' previous state in valid and unique way

Integrating with Unicity Tokens

i.e. infra for 2-way trust without running a full node of another

- EVM contracts must be able to validate Unicity Tokens
  - there must be a validated root of trust - we implement it as a system contract (builtin) which can validate unicity certificates (alternative: EIP-4788 based something)
  - there must be Solidity library validating full tokens, including their mint reasons, etc.
- Token layer must be able to validate EVM artifacts
  - we include Unicity Certificates certifying blocks into EVM blocks, so that everyone can extract their interesting transaction or receipts or logs/events together with hash chain to UC

Execution loop

- L2 zk-rollups cheat: they provide centralized execution and eventual decentralized verification.
    1. sequence a block
    2. execute
    3. and return soft finality to users
    4. eventually prove a batch of blocks
    5. finally submit batched proof to L1 contract
    6. after L1 inclusion is confirmed the finality is final

(proving costs and L1 fees amortize)

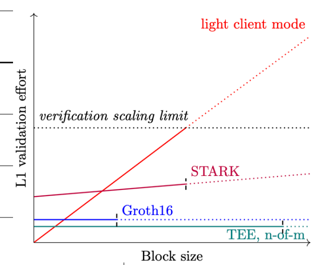
- We do it synchronously with real finality
    1. sequence a block
    2. execute
    3. prove
    4. submit proof to BFT Core
    5. receive UC, finalize the block

(next block processing may start optimistically when proving starts)

Different kinds of proofs

trust and scaling

	trust (evm layer)	scaling
no proof	honest EVM machine but no forking, no backdating	awesome, log
EVM is a cluster of machines	honest simple majority	shardable, log
ZK	crypto works (trusted setup)	slow proving, constant time verification, shardable
BFT Core in light client mode	awesome	linear, sharding up to a certain hard limit
TEE	Hardware + Intel + attestation (+ sysadmin?)	awesome in theory



Different kinds of proofs
properties

	proof size	proving effort	BFT Core's verification effort
Compressed (recursive STARK, we're using this)	1.5MB	**hard**	light
Core (STARKs)	Grows linearly	a bit less hard	light, but more bandwidth overhead
Compressed + groth16 (standard for Eth L1)	200 bytes	**hard** + then hard, and trusted setup	lightest
TEE	n/a	just run in attested vm	pray
BFT Core in light client mode	block's **txs** + touched accounts, code	just execution	linear with **hard limit** (thousands of tps)
n-of-m cluster	m * n/a	m * execution	n * messages

zk proving speed

- 2026-01-08T19:46:37.342421Z INFO Block 4 details: 1 transactions, 0 ommers, gas_used: 21000, gas_limit: 30000000
- 2026-01-08T19:46:37.343256Z INFO Generated execution witness for block 4 (0 codes, 3 keys)
- 2026-01-08T19:51:28.949423Z INFO Generated proof for block 4 (1477450 bytes)
- ...
- 2026-01-08T19:53:32.516745Z INFO Block 5 details: 5 transactions, 0 ommers, gas_used: 105000, gas_limit: 30000000
- 2026-01-08T19:53:32.517877Z INFO Generated execution witness for block 5 (0 codes, 3 keys)
- 2026-01-08T19:58:50.393142Z INFO Generated proof for block 5 (1477450 bytes)

CPU	> 5 min	on my machine	scales linearly with extra hw
GPU	1 min?	needs high-end GPUs	example -->
Prover Network	15 sec?	10..50 cents per block?	

Block (mainnet)	Gas Used	ethrex (SP1 Turbo 1×4090)	ethrex (ZisK 0.14.0 1×4090)	ethrex (ZisK 0.14.0 1×5090)	ethrex (ZisK 0.14.0 16×5090)
23769082	7,949,562	02m 23s	58s	33s	6s
23769083	44,943,006	12m 24s	5m	3m 57s	23s 600ms

BFT Core in Light Client Mode

- Prover (proves that running the "program", with those inputs, returns OK):

```
(proof, public_inputs)  <-- ZKProve(program, public_inputs, secret_inputs)
```

- Verifier:

```
valid/not <-- ZKVerify(program_id, (proof, public_inputs))
// if valid then public_inputs are now validated and ready for further checks
```

Observing that the tooling for ZK proving generates awesome self-contained light client inputs we can do the following instead:

- Prover: identity (does nothing)
- Verifier:

```
valid/not <-- program(public_inputs, secret_inputs)
// if valid then all inputs are valid and we continue with checks based on pub
```

- This is more efficient to validate than ZK until 1000 tps perhaps? With zero proving effort.

Inputs, specifically:

- Public inputs:
  - chain id
  - fork id (enough for "versioning")
  - block number (ignored)
  - previous block hash (ignored)
  - block hash
  - previous state root hash
  - state root hash
  - gas used (ignored now)
- Secret inputs:
  - transactions, receipts
  - account witnesses (pre state, MPT proof)
  - storage witnesses (contract id, slot pre value, MPT proof)
  - creation/deletion witnesses
  - execution witness code (bytecode)
  - helper indexes, etc for efficiency

Implementation

- https://github.com/ristik/uni-evm
- https://github.com/unicitynetwork/bft-core/tree/l1
-

```
                    Uni-EVM (Rust)

  Block Producer → Proof Coordinator → BFT Committer
       ↓                  ↓                  ↓
  Execute Block   Generate SP1 Proof   Submit to L1
                  (ethrex backend)

                          ↓
              BlockCertificationRequest
              - InputRecord (prev_hash, new_hash)
              - ZK Proof (~1.5MB bincode-serialized)
              - Signature (secp256k1)
                          ↓

                 BFT-Core (Go + Rust FFI)

  Certification Handler → ZK Verifier → Consensus
          ↓                    ↓              ↓
   Receive Request    Verify SP1 Proof  Issue UC
                      (FFI → Rust lib)

                          ↓
                  UnicityCertificate
                  - InputRecord (certified)
                  - UnicitySeal (signatures)
                  - TechnicalRecord (next_round)
```

Implementation: code

```
• uni-evm/
• ├── crates/
• │   ├── uni-bft-committer/    # BFT Core integration, libp2p/CBOR/messaging
• │   ├── uni-bft-precompile/   # EVM precompile for Unicity Certificate validation
• │   ├── uni-sequencer/        # Block production + SP1 proof coordination
• │   └── uni-storage/          # Storage (UCs + proofs)
• ├── cmd/uni-evm/             # Main binary (node orchestration + RPC)
• ├── guest-program/           # runs in zkVM, "light client" validation
• └── tools/extract-vkey/      # extracts guest program's cryptographic id

• bft-core/
• ├── rootchain/node.go                 # includes proof verifiation step if enabled
• ├── rootchain/consensus/zkverifier/    # verifier in go, calls,,
• └── rootchain/consensus/zkverifier/sp1-verifier-ffi/   # rust project root (links to sp1)
```

Interfaces in BFT-Core

```go
type BlockCertificationRequest struct {
    _           struct{}            `cbor:",toarray"`
    PartitionID types.PartitionID   `json:"partitionId"`
    ShardID     types.ShardID       `json:"shardId"`
    NodeID      string              `json:"nodeId"`
    InputRecord *types.InputRecord  `json:"inputRecord"`
    ZkProof     []byte              `json:"zkProof"` // ZK proof for state transition validation
    BlockSize   uint64              `json:"blockSize"`
    StateSize   uint64              `json:"stateSize"`
    Signature   hex.Bytes           `json:"signature"`
}
```

```go
// ZK verification flags
cmd.Flags().BoolVar(&flags.ZKVerificationEnabled, "zk-verification-enabled", false,
    "Enable ZK proof verification for L2 state transitions")
cmd.Flags().StringVar(&flags.ZKProofType, "zk-proof-type", "sp1",
    "ZK proof type (sp1, risc0, exec, none)")
cmd.Flags().StringVar(&flags.ZKVerificationKeyPath, "zk-vkey-path", "",
    "Path to ZK verification key file (.vkey)")
```

```go
// verifyZKProof verifies the ZK proof in the block certification request
func (v *Node) verifyZKProof(ctx context.Context, req *certification.BlockCertificationRequest, si *storage.ShardInfo)
```

Proof Verification in Rust

See:

/rootchain/consensus/zkverifier/sp1-verifier-ffi/README.md