Engineering EVM Precompiles for Cryptographic Primitives

Zach Kelling

Lux Industries

2025

Abstract

An EVM precompile is a native function callable from Solidity at a designated address, executing outside the EVM interpreter at near-native speed. Precompiles enable cryptographic operations that would be prohibitively expensive in EVM bytecode. This paper teaches the engineering of precompiles from first principles: the EVM precompile interface, address space conventions, input/output ABI design, gas cost modeling based on empirical benchmarks, determinism requirements, error handling, and testing methodology. We present all 20 precompiles deployed on Lux Network's C-Chain, covering BLS12-381, BN254, batch secp256k1, post-quantum signatures (ML-DSA, FALCON), FHE operations, Warp message verification, and NTT. For each precompile, we provide the Go implementation, gas cost analysis, security considerations, and Solidity interface contract. This is a complete guide for blockchain engineers adding native cryptographic capabilities to EVM-compatible chains.

Contents

1 What Is a Precompile

1.1 The EVM Execution Model

The EVM executes smart contracts as sequences of opcodes (bytecodes). Each opcode has a fixed gas cost. Opcodes provide basic operations: arithmetic, memory access, storage, control flow. There are no opcodes for complex cryptographic operations like pairing checks or post-quantum signature verification.

1.2 Precompiles: Native Extensions

A precompile is a special contract address (typically in the range `0x0001`–`0x00ff` for Ethereum, `0x0200`–`0x02ff` for Lux extensions) where a `CALL` or `STATICCALL` invokes a native Go function instead of interpreting EVM bytecode.

From Solidity's perspective, a precompile call looks like a normal external call:

Listing 1: Calling a precompile from Solidity

```
// Call the BLS12-381 pairing precompile at 0x0200
(bool success, bytes memory result) = address(0x0200).staticcall(
    abi.encodePacked(g1Point, g2Point)
);
require(success, "pairing check failed");
bool pairingResult = abi.decode(result, (bool));
```

1.3 Why Precompiles Instead of Libraries

Implementing BLS12-381 pairing in Solidity would cost approximately 50M gas (most of the block gas limit). The same operation as a precompile costs 113,000 gas—a 440x reduction. The reason: EVM opcodes have overhead per operation (gas metering, stack manipulation, memory management), and cryptographic operations require millions of field multiplications that incur this overhead for each one.

2 The Precompile Interface

2.1 Go Interface

Every precompile implements a standard Go interface:

Listing 2: The precompile interface

```
// PrecompiledContract is the interface for native contracts.
type PrecompiledContract interface {
    // RequiredGas returns the gas cost for the given input.
    // Must be deterministic: same input -> same gas.
    RequiredGas(input []byte) uint64

    // Run executes the precompile with the given input.
    // Returns output bytes and an error.
    // Must be deterministic: same input -> same output.
    Run(input []byte) ([]byte, error)
}
```

Two methods, both deterministic:

1. `RequiredGas`: Called before execution to check if the caller has enough gas. Must not perform expensive computation—it should be $O(1)$ or $O(\text{input length})$.

2. `Run`: Performs the actual computation. Returns output bytes or an error. On error, all gas is consumed (precompile failure is an out-of-gas condition).

2.2 Registration

Precompiles are registered in the VM configuration:

Listing 3: Registering a precompile

```
var PrecompiledContractsLux = map[common.Address]PrecompiledContract{
    // Standard Ethereum precompiles (0x01 - 0x09)
    common.BytesToAddress([]byte{0x01}): &ecRecover{},
    common.BytesToAddress([]byte{0x02}): &sha256hash{},
    // ...

    // Lux extensions (0x0200 - 0x02ff)
    common.BytesToAddress([]byte{0x02, 0x00}): &bls12381Pairing{},
    common.BytesToAddress([]byte{0x02, 0x01}): &bls12381G1Add{},
    common.BytesToAddress([]byte{0x02, 0x02}): &bls12381G1Mul{},
    common.BytesToAddress([]byte{0x02, 0x03}): &bls12381MapFp{},
    common.BytesToAddress([]byte{0x02, 0x10}): &bn254Pairing{},
    common.BytesToAddress([]byte{0x02, 0x20}): &batchEcRecover{},
    common.BytesToAddress([]byte{0x02, 0x30}): &dilithiumVerify{},
    common.BytesToAddress([]byte{0x02, 0x31}): &falconVerify{},
    common.BytesToAddress([]byte{0x02, 0x40}): &fheEncrypt{},
    common.BytesToAddress([]byte{0x02, 0x41}): &fheAdd{},
    common.BytesToAddress([]byte{0x02, 0x42}): &fheMul{},
    common.BytesToAddress([]byte{0x02, 0x50}): &warpVerify{},
    common.BytesToAddress([]byte{0x02, 0x60}): &nttForward{},
    common.BytesToAddress([]byte{0x02, 0x61}): &nttInverse{},
    // ... (20 total)
}
```

2.3 Address Space Convention

Range	Owner	Purpose
`0x0001`–`0x0009`	Ethereum	Standard precompiles
`0x000a`–`0x00ff`	Ethereum	Reserved for EIPs
`0x0100`–`0x01ff`	Reserved	Chain-specific
`0x0200`–`0x020f`	Lux	BLS12-381 operations
`0x0210`–`0x021f`	Lux	BN254 extensions
`0x0220`–`0x022f`	Lux	Batch operations
`0x0230`–`0x023f`	Lux	Post-quantum signatures
`0x0240`–`0x024f`	Lux	FHE operations
`0x0250`–`0x025f`	Lux	Warp/cross-chain
`0x0260`–`0x026f`	Lux	Polynomial/NTT
`0x9000`–`0x90ff`	Lux	Application adapters (DEX)

Table 1: Precompile address space allocation. The `0x90xx` application range holds non-cryptographic adapters; the canonical DEX CLOB receipt-settlement adapter at `0x9999` (the deprecated `0x9010` aliases to it) is documented in Section 11.

3 Input/Output ABI

3.1 Design Principles

1. **Fixed-size fields**: All inputs use fixed-width encoding (32 bytes for scalars, 48 bytes for G1 points, etc.). No length-prefixed variable fields.
2. **Left-padded**: Scalars are left-padded with zeros to 32 bytes, matching Solidity ABI conventions.
3. **Concatenated**: Multiple inputs are concatenated without separators. The precompile deduces structure from the total input length.
4. **Error = revert**: Invalid inputs cause the precompile to return an error, which the EVM interprets as a revert consuming all provided gas.

3.2 Example: BLS12-381 Pairing Check

Listing 4: BLS12-381 pairing input/output format

```
// Input: k pairs of (G1, G2) points, concatenated
// Each pair: 128 bytes (48 for G1 compressed + 96 for G2 compressed,
//            padded to 128 for alignment)
// Total input: k * 128 bytes
//
// Output: 32 bytes
// 0x00...01 if pairing check passes (product of pairings == 1)
// 0x00...00 if pairing check fails

func (c *bls12381Pairing) Run(input []byte) ([]byte, error) {
    if len(input) == 0 || len(input)%128 != 0 {
        return nil, errInvalidInput
    }

    k := len(input) / 128
    pairs := make([]PairingPair, k)

    for i := 0; i < k; i++ {
        chunk := input[i*128 : (i+1)*128]

        g1, err := decodeG1(chunk[:48])
        if err != nil {
            return nil, err
        }
        if !g1.IsOnCurve() || !g1.IsInSubgroup() {
            return nil, errNotOnCurve
        }

        g2, err := decodeG2(chunk[48:128])
        if err != nil {
            return nil, err
        }
        if !g2.IsOnCurve() || !g2.IsInSubgroup() {
            return nil, errNotOnCurve
        }

        pairs[i] = PairingPair{G1: g1, G2: g2}
    }

    result := multiPairing(pairs)
```

```
    output := make([]byte, 32)
    if result.IsOne() {
        output[31] = 1
    }
    return output, nil
}
```

4 Gas Cost Modeling

4.1 Principle: Gas Reflects Wall-Clock Time

Gas costs must reflect actual computation time to prevent denial-of-service attacks. If a precompile is underpriced, an attacker can consume node CPU without paying proportional gas. If overpriced, legitimate use is discouraged.

4.2 Calibration Method

1. Benchmark the precompile on reference hardware (AMD EPYC 7713, which runs Lux Network validators).
2. Measure wall-clock time for representative inputs.
3. Compute gas per nanosecond using the baseline: `ecrecover` costs 3,000 gas and takes approximately 26 μs on reference hardware, giving a rate of approximately 0.115 gas/ns.
4. Gas cost = execution time (ns) $\times$ 0.115, rounded up to the nearest 1,000.

4.3 Gas Cost Table

Precompile	Time (μs)	Gas	Gas/ns
BLS12-381 pairing (1 pair)	1,700	113,000	0.066
BLS12-381 G1 add	2.4	500	0.208
BLS12-381 G1 scalar mul	340	45,000	0.132
BN254 pairing (1 pair)	820	65,000	0.079
Batch ecrecover (100 sigs)	12,000	180,000	0.015
ML-DSA-65 verify	300	35,000	0.117
FALCON-512 verify	180	22,000	0.122
FHE encrypt	450	55,000	0.122
FHE add	12	2,000	0.167
FHE multiply	850	100,000	0.118
Warp message verify	95	12,000	0.126
NTT-256 forward	3.1	500	0.161
NTT-256 inverse	3.3	500	0.152

Table 2: Gas costs calibrated to execution time on AMD EPYC 7713.

All gas/ns ratios fall within the range $[0.015, 0.208]$, a factor of 14x variation. The target is 0.115 ± 2x. Batch ecrecover is intentionally underpriced to incentivize batching (it replaces 100 individual ecrecover calls at 300,000 gas with 180,000 gas for the batch).

5 Determinism Requirements

5.1 Why Determinism Is Non-Negotiable

Every validator must compute identical state transitions. If a precompile produces different outputs on different hardware, validators disagree on the state root, causing a consensus split.

5.2 Sources of Non-Determinism

1. **Floating-point arithmetic**: Never use floating point in precompiles. All arithmetic is integer or modular.
2. **Map iteration order**: Go map iteration is randomized. Never iterate over maps in precompile code.
3. **Goroutines**: Concurrent execution with shared state is non-deterministic. Precompiles must be single-threaded.
4. **System calls**: Time, random number generation, file I/O. Never use these.
5. **Compiler optimizations**: Different Go compiler versions may produce different results for certain edge cases. Pin the Go version and test across platforms.

5.3 Determinism Testing

We run every precompile test on three platforms (AMD64, ARM64, WASM) and verify identical outputs. Any divergence is a critical bug.

Listing 5: Determinism test

```
func TestDeterminism(t *testing.T) {
    // Golden outputs computed on reference platform
    for _, tc := range goldenTestCases {
        output, err := precompile.Run(tc.Input)
        if err != nil {
            t.Fatalf("unexpected␣error:␣%v", err)
        }
        if !bytes.Equal(output, tc.ExpectedOutput) {
            t.Fatalf("determinism␣violation:\n"+
                "input:␣␣␣␣%x\n"+
                "got:␣␣␣␣␣␣%x\n"+
                "expected:␣%x",
                tc.Input, output, tc.ExpectedOutput)
        }
    }
}
```

6 Error Handling

6.1 Precompile Error Semantics

When a precompile returns an error, the EVM:

1. Consumes all gas provided to the call.
2. Returns no output data.
3. Reverts state changes made within the call (but the gas is still consumed).

6.2 Input Validation

Every precompile validates inputs before processing:

Listing 6: Input validation pattern

```
func (c *dilithiumVerify) Run(input []byte) ([]byte, error) {
    // Check total length
    if len(input) < PubKeyLen+SigLen+32 {
        return nil, errInvalidInputLength
    }

```

```
    // Decode public key
    pk, err := decodeDilithiumPubKey(input[:PubKeyLen])
    if err != nil {
        return nil, errInvalidPublicKey
    }

    // Decode signature
    sig, err := decodeDilithiumSig(input[PubKeyLen : PubKeyLen+SigLen])
    if err != nil {
        return nil, errInvalidSignature
    }

    // Message is the remainder
    msg := input[PubKeyLen+SigLen:]

    // Verify
    valid := dilithium.Verify(pk, msg, sig)

    output := make([]byte, 32)
    if valid {
        output[31] = 1
    }
    return output, nil
}
```

6.3 Denial-of-Service Resistance

The `RequiredGas` function must accurately bound the computation cost. For variable-cost operations (e.g., pairing with k pairs), the gas scales linearly:

Listing 7: Variable gas cost for pairing

```
func (c *bls12381Pairing) RequiredGas(input []byte) uint64 {
    if len(input) == 0 || len(input)%128 != 0 {
        return 0 // Will fail in Run
    }
    k := uint64(len(input)) / 128
    return 43000 + k*70000 // Base + per-pair cost
}
```

7 The 20 Precompiles

Addr	Name	Gas	Function
0x0200	BLS12-381 Pairing	113K/pair	Multi-pairing check
0x0201	BLS12-381 G1 Add	500	Elliptic curve addition
0x0202	BLS12-381 G1 Mul	45K	Scalar multiplication
0x0203	BLS12-381 Map-to-G1	5.5K	Hash to curve
0x0204	BLS12-381 G2 Add	800	G2 point addition
0x0205	BLS12-381 G2 Mul	128K	G2 scalar multiplication
0x0206	BLS12-381 Map-to-G2	23K	Hash to G2
0x0207	BLS12-381 Aggregate	12K	Aggregate BLS signatures
0x0210	BN254 Pairing	65K/pair	BN254 multi-pairing
0x0211	BN254 G1 Mul Batch	6K/point	Batched scalar mul
0x0220	Batch ecrecover	1.8K/sig	Batch secp256k1 verify
0x0230	ML-DSA Verify	35K	Dilithium-3 verification
0x0231	FALCON Verify	22K	FALCON-512 verification
0x0240	FHE Encrypt	55K	TFHE encryption
0x0241	FHE Add	2K	Ciphertext addition
0x0242	FHE Multiply	100K	Ciphertext multiplication
0x0243	FHE Decrypt	45K	Ciphertext decryption
0x0250	Warp Verify	12K	Cross-chain message verify
0x0260	NTT Forward	500	Forward NTT-256
0x0261	NTT Inverse	500	Inverse NTT-256

Table 3: All 20 Lux Network precompiles with address and gas cost.

8 Testing Methodology

8.1 Four-Level Testing

1. **Known-Answer Tests (KATs)**: Test against NIST/standard test vectors. Verifies correctness for specific inputs.

2. **Fuzz Tests**: Random inputs to find panics, incorrect results, and non-determinism.

3. **Integration Tests**: Call precompiles from Solidity via Foundry. Verifies the full path from Solidity through the EVM to the native code and back.

4. **Cross-Platform Tests**: Run identical tests on AMD64, ARM64, and WASM. Any output difference is a critical bug.

Listing 8: Integration test for post-quantum verification

```
function test_dilithiumVerifyFromSolidity() public {
    // Known-good signature from NIST test vectors
    bytes memory input = abi.encodePacked(
        dilithiumPubKey,
        dilithiumSignature,
        testMessage
    );

    (bool success, bytes memory result) = address(0x0230).staticcall(
       input);
    assertTrue(success, "precompile call failed");

    uint256 verified = abi.decode(result, (uint256));
```

```
    assertEq(verified, 1, "valid signature should verify");
}

function test_dilithiumRejectsInvalidSig() public {
    bytes memory invalidSig = new bytes(dilithiumSignature.length);
    // Zero signature should not verify
    bytes memory input = abi.encodePacked(
        dilithiumPubKey,
        invalidSig,
        testMessage
    );

    (bool success, bytes memory result) = address(0x0230).staticcall(
       input);
    assertTrue(success, "precompile should not revert on invalid sig");

    uint256 verified = abi.decode(result, (uint256));
    assertEq(verified, 0, "invalid signature should not verify");
}
```

9 Security Considerations

9.1 Subgroup Checks

For BLS12-381 and BN254, input points must be checked for subgroup membership. Accepting points outside the prime-order subgroup enables small-subgroup attacks that can forge signatures or break pairing assumptions.

9.2 Gas Griefing

If `RequiredGas` underestimates the true cost, an attacker can craft inputs that consume more CPU time than the gas price reflects, slowing down validators. We benchmark worst-case inputs (maximum-cost curve operations, maximum-length batch operations) and set gas costs to the worst case.

9.3 Timing Side Channels

Precompile execution time must not depend on secret data. For verification precompiles, this is straightforward: the input (public key, signature, message) is public. For FHE precompiles that handle encrypted data, we use constant-time implementations.

10 Solidity Interface Contracts

We provide Solidity libraries for type-safe precompile access:

Listing 9: Solidity interface library

```
library PQCrypto {
    address constant DILITHIUM = address(0x0230);
    address constant FALCON = address(0x0231);

    function verifyDilithium(
        bytes memory pubKey,
        bytes memory signature,
        bytes memory message
```

```
    ) internal view returns (bool) {
        (bool ok, bytes memory result) = DILITHIUM.staticcall(
            abi.encodePacked(pubKey, signature, message)
        );
        if (!ok || result.length != 32) return false;
        return abi.decode(result, (uint256)) == 1;
    }

    function verifyFalcon(
        bytes memory pubKey,
        bytes memory signature,
        bytes memory message
    ) internal view returns (bool) {
        (bool ok, bytes memory result) = FALCON.staticcall(
            abi.encodePacked(pubKey, signature, message)
        );
        if (!ok || result.length != 32) return false;
        return abi.decode(result, (uint256)) == 1;
    }
}
```

11 The DEX Receipt-Settlement Precompile (0x9999)

The twenty precompiles documented above are *cryptographic* primitives: pure functions of their input bytes with no external state. Lux adds one precompile of a different kind — an *application adapter* — at address `0x9999`, in the application range `0x9000`–`0x90ff` reserved above the cryptographic block. It is the canonical EVM-side entry point to the Lux DEX, and its engineering is instructive precisely because it is not a pure function: it is a deterministic *settlement* adapter over a separate matching chain. The earlier `0x9010` adapter is deprecated and retired to a single namespace-shared path; `0x9999` is the sole production money path. The normative specification is LP-9999 [8]; this section documents the EVM-side engineering and defers the full settlement contract to it.

11.1 A CLOB Settlement Adapter, Not an AMM and Not a Relay

`0x9999` exposes the Uniswap-V4-`PoolManager` ABI unchanged — pool keys, hooks, flash accounting — but it is neither an automated market maker nor a live matcher. A call does not move along a bonding curve, and — crucially — it does *not* relay an order to a matching engine and wait for a fill. The matching engine, the resting book, and value conservation live entirely on the **D-Chain** (the Lux Lightspeed DEX matching VM [7]). The D-Chain matches an order and a D-validator quorum BLS-signs a `DFillReceipt`; `0x9999` then *settles that receipt* — it verifies the certificate and debits/credits EVM balances. The V4-shaped surface is retained so that existing V4-aware routers and Solidity integrators compose unchanged, while the execution semantics underneath are price-time priority CLOB matching certified out-of-band rather than constant-product swaps.

This is deliberately distinct from the plain Solidity AMMs Lux also ships: the V2 (constant-product) and V3 (concentrated-liquidity) pools are ordinary contracts with no precompile involvement. The precompile exists only for the CLOB path, where on-chain bytecode cannot match a limit-order book at the throughput the D-Chain achieves natively. The design slogan is *D matches · C settles · BLS certifies day-1 · Q later · X finalizes.*

11.2 The Build-vs-Verify Split (Why Not a Live Relay)

A live relay is fork-unsafe. If the precompile called a matcher against a moving book during block execution, every validator would re-execute that call at *Verify* at a different instant, obtain a different fill, and compute a different state root — a consensus split. The resolution is a strict **build-vs-verify split** carried by a certificate. On the *build/propose* path, a node that validates the D-Chain obtains the fill from its local matcher and assembles a signed `DFillReceipt`. On the *verify/replay* path — every C-Chain validator, inside `Run` — the precompile MUST NOT query any live matcher; it settles solely from the certificate carried in the V4 `hookData` (or, equivalently, the block-header predicate rail). Because certificate verification is a pure function of the receipt bytes and the verifier key, it yields an identical result on every validator. This is exactly the determinism contract every other precompile in this paper honours — here achieved not by statelessness but by verifying a certificate rather than recomputing a fill.

11.3 Inline BLS Certificate Verification (Deterministic)

The day-1 certificate is a BLS aggregate signature over a receipt root: a D-validator quorum signs the per-block (or per-batch) receipt root, and each C-Chain swap carries the receipt, its Merkle inclusion proof, and one BLS certificate. Aggregate BLS verification is deterministic and side-effect free, so it is safe to perform inside `Run` — the same pairing machinery documented in the cryptographic sections above, applied to an application certificate. The certificate kind is a versioned `certType` enum resolved through an on-chain verifier registry, so the cryptography upgrades with **no ABI change**: BLS fast-path on day-1, then QuasarCert, a post-quantum finality profile, or a STARK/zk batch proof register a new `certType` at an activation height while the V4 ABI and the `DFillReceipt` envelope stay fixed.

11.4 Native LUX as `address(0)`

The pool-key encoding follows the V4 convention that the native asset is the zero address. A pool key referencing `address(0)` denotes native LUX; ERC-20 securities use their token address. This removes the wrapped-native special case from the adapter ABI: every asset, native or token, is an address in the pool key, and `address(0)` routes to the D-Chain's native-balance path. Internally, settlement binds the full 32-byte `AssetID` — never an alias or fold — so the pool-key address is resolved to a canonical asset identity before any balance moves.

11.5 Exactly-Once Settlement via Consumed Receipts

A settlement precompile faces a hazard a pure precompile does not: the same EVM call may be re-executed (block re-org, validator restart, speculative execution) and must not settle the same fill twice. Each `DFillReceipt` carries a unique `receiptID`, and the precompile records `consumedReceipt[receiptID]` on success; re-presenting a consumed receipt reverts. The receipt is further bound to exactly one swap on exactly one chain: it commits to the pool-key hash, the swap-parameters hash, the sender, the recipient, the minimum output, the deadline, and the precompile address `0x9999`, and the precompile reverts on any mismatch. A revert means nothing happened — no input taken, no output credited, no partial DEX state — preserving the all-or-nothing ownership invariant that only the named counterparties move value. Replay protection is thus state (the consumed-receipt set) rather than content-addressing, which is what makes settlement, as opposed to a fire-and-forget relay, safe and idempotent.

11.6 Block-STM Parallel Settlement

`0x9999` is Block-STM-aware. It declares a fine-grained read/write access set — the consumed-receipt slot for this receipt, the sender's and recipient's per-asset balance slots, the relevant

pool and asset registry entries, the verifier-registry entry for the signing validator set, and a sharded fee bucket — with **no global hot write slots** (no global nonce, volume counter, single fee accumulator, or last-receipt slot). Two swaps conflict only if they share an account, asset, receipt, allowance, or pool key, so the universal C-Chain validator set becomes a parallel settlement fabric: thousands of independent fills settle concurrently and only genuine conflicts serialize. Trading is stoppable at fine-grained halt scopes (global, per-market, per-asset, per-receipt-type, per-validator-set), and the fund-preserving default on an incident is to stop new swaps while still allowing cancels, settlement of already-certified safe receipts, and withdrawals.

11.7 Why This Belongs in the Precompile Layer

A CLOB cannot be matched in EVM bytecode at competitive cost — the same argument that justifies the cryptographic precompiles — and a BLS aggregate certificate cannot be verified in bytecode at competitive gas. The adapter keeps the EVM as the settlement and composition surface (V4-compatible, hook-friendly) while delegating latency-critical matching to a chain engineered for it and certificate verification to the precompile layer. The full D-Chain VM architecture — matcher-at-*Verify*, `zapdb` commit, crash-restart durability, and the GPU CLOB matcher behind it — is specified in the Lux Lightspeed DEX paper [7], and the receipt-settlement contract — receipt binding, the certType verifier registry, the Block-STM access set, halt scopes, and the `0x9010` migration — is normative in LP-9999 [8]. This section documents only the EVM-side adapter and the engineering discipline (build-vs-verify split, `address(0)` native, inline deterministic certificate verification, exactly-once consumed receipts) that makes a stateful settlement adapter safe to expose as a precompile.

12 Conclusion

EVM precompiles are the mechanism by which blockchain VMs gain new cryptographic capabilities without changing the EVM instruction set. Engineering them correctly requires attention to determinism, gas pricing, input validation, error handling, and cross-platform testing. Done well, precompiles enable practical use of advanced cryptography (post-quantum signatures, FHE, pairing-based proofs) in smart contracts at gas costs comparable to basic operations.

Our 20 precompiles provide Lux Network's C-Chain with a comprehensive cryptographic toolkit. Each one follows the same engineering discipline: deterministic execution, gas cost calibrated to wall-clock time, subgroup and input validation, and four levels of testing.

Canonical c-abi migration (cevm v0.46.0–v0.47.0)

cevm v0.46.0 (Phase 5b) rewired `cevm/lib/cevm_precompiles/bls.cpp` and `kzg.cpp` (564 lines of in-tree blst) into thin `extern "C"` wiring through the brand-neutral `luxcpp/crypto/bls` and `luxcpp/crypto/kzg` c-abi (EIP-2537 BLS12-381 G1/G2 add/mul/msm + pairing_check + map_fp/fp2, EIP-4844 KZG point-evaluation). Production binaries (`libcevm_precompiles.a`, `libevm.dylib`, `libevm-kernel-metal.a`) link zero blst symbols, asserted in CI by `cevm/test/unittests/no_bl` on every build (`nm` + `otool -L` + `nm -a` all clean). cevm v0.47.0 (commit `aea6ba08`) migrated 7 additional algorithms to forward-shim headers calling the canonical luxcpp/crypto c-abi, removing 2 107 lines of duplicated implementation. Pairing plus keccak / secp256k1 are deferred pending relative-include flatten and legacy API consumer migration. The subgroup policy (`SubgroupPolicy::CheckAndReject` for validator pubkeys, `ClearIfHashToCurve` only for hash-to-curve outputs) lives in the canonical c-abi and is exercised against blst as a test-only oracle pinned at `luxcpp/crypto/bls/test/cmake/`.

References

[1] Reitwiessner, C. "EIP-196: Precompiled Contracts for Addition and Scalar Multiplication on the Elliptic Curve alt_bn128." 2017.

[2] Reitwiessner, C. "EIP-197: Precompiled Contracts for Optimal Ate Pairing Check on the Elliptic Curve alt_bn128." 2017.

[3] Vlasov, A. et al. "EIP-2537: Precompile for BLS12-381 Curve Operations." 2020.

[4] Wood, G. "Ethereum: A Secure Decentralised Generalised Transaction Ledger (Yellow Paper)." 2014.

[5] NIST. "Module-Lattice-Based Digital Signature Standard (ML-DSA)." FIPS 204, 2024.

[6] Prest, T. et al. "FALCON: Fast-Fourier Lattice-Based Compact Signatures over NTRU." NIST PQC, 2020.

[7] Lux Industries, Inc. "Lux Lightspeed DEX: On-Chain Order Book with Sub-Second Finality." Lux Technical Report.

[8] Lux Industries, Inc. "LP-9999: DEX V4 Receipt-Settlement Precompile (`0x9999`)." Lux Proposal (Standards Track, Core). https://github.com/luxfi/lps/blob/main/LPs/lp-9999-dex-v4-receipt-settlement-precompile.md.