Custom EVM Precompiles for Cryptographic Operations on Lux Network

BLS12-381, BN254, Batch Verification, Post-Quantum Signatures, and Gas Cost Analysis for Next-Generation Smart Contracts

Zach Kelling

Lux Industries

Original: 2021 | Revised: February 2026

Abstract

We present a suite of custom EVM precompiled contracts deployed on Lux Network's C-Chain and EVM-compatible appchains, providing native-speed cryptographic operations that would be prohibitively expensive if implemented in Solidity. The precompile suite includes: (i) full BLS12-381 pairing operations for aggregate signature verification and zero-knowledge proof systems; (ii) BN254 (alt_bn128) extensions for efficient Groth16 verification; (iii) secp256k1 batch signature verification achieving 5x throughput improvement over sequential `ecrecover`; (iv) a post-quantum signature verification precompile supporting ML-DSA (FIPS 204, formerly CRYSTALS-Dilithium) and FALCON for quantum-resistant authentication; and (v) a Warp message verification precompile that validates cross-appchain BLS aggregate proofs in a single call. We provide detailed gas cost models calibrated to actual execution time on reference hardware, ensuring that gas prices reflect computational cost within a factor of 2. Benchmarks demonstrate that BLS12-381 pairing verification completes in 1.7ms (113,000 gas), Dilithium-3 verification in 0.3ms (35,000 gas), and batch verification of 100 secp256k1 signatures in 12ms (180,000 gas versus 300,000 gas for 100 sequential `ecrecover` calls). All precompiles are assigned addresses in the `0x0200...` namespace to avoid conflicts with Ethereum's precompile address space.

1 Introduction

The Ethereum Virtual Machine (EVM) provides a limited set of precompiled contracts for cryptographic operations: `ecrecover` (secp256k1 signature recovery), SHA-256, RIPEMD-160, identity, modular exponentiation, and BN254 curve operations (EIP-196/197) [1, 2, 3]. While these suffice for basic Ethereum functionality, advanced applications—zero-knowledge proofs, aggregate signatures, cross-chain verification, and post-quantum cryptography—require operations that are either absent from the standard precompile set or prohibitively expensive when implemented in Solidity bytecode.

Lux Network addresses this gap through custom precompiles: native Go functions callable from Solidity at designated addresses, executing outside the EVM interpreter at near-native speed. This approach is possible because Lux appchains control their own VM implementations, allowing protocol-level extensions without Ethereum consensus.

This paper makes the following contributions:

(i) A comprehensive precompile suite covering five cryptographic families with formal interface specifications.

(ii) A gas pricing model based on empirical benchmarking that ensures gas costs reflect wall-clock execution time within a constant factor.

(iii) Security analysis of each precompile, including input validation, error handling, and denial-of-service resistance.

(iv) Comparison with EIP-2537 (BLS12-381) and other proposed Ethereum precompiles, showing that Lux's implementation achieves lower gas costs through tighter integration.

(v) Post-quantum readiness analysis demonstrating practical ML-DSA (FIPS 204, formerly CRYSTALS-Dilithium) and FALCON verification on-chain.

2 Precompile Architecture

2.1 Address Space

Lux precompiles occupy the address range `0x0200000000000000000000000000000000000001` through `0x02000000000000000000000000000000000000FF`, avoiding collision with Ethereum precompiles (`0x01`–`0x09`) and proposed precompiles (`0x0A`–`0x13`).

Table 1: Lux custom precompile address assignments.

Address (suffix)	Precompile	Gas Base
0x01	BLS12-381 G1 Add	500
0x02	BLS12-381 G1 Mul	12,000
0x03	BLS12-381 G1 Multi-Exp	12,000+
0x04	BLS12-381 G2 Add	800
0x05	BLS12-381 G2 Mul	45,000
0x06	BLS12-381 Pairing	113,000+
0x07	BLS12-381 Map to G1	5,500
0x08	BLS12-381 Map to G2	110,000
0x10	BN254 Extended Pairing	80,000+
0x11	BN254 Batch Pairing	45,000+
0x20	secp256k1 Batch Verify	3,000+
0x30	Dilithium-3 Verify	35,000
0x31	FALCON-512 Verify	28,000
0x32	FALCON-1024 Verify	42,000
0x40	Warp Message Verify	50,000
0x41	SHA3-256 (Keccak)	30+
0x42	BLAKE2b-256	25+

2.2 Calling Convention

All precompiles follow the standard EVM precompile calling convention:

- Input: ABI-encoded arguments passed via `STATICCALL` or `CALL`.
- Output: ABI-encoded result written to return data.
- Gas: Deducted before execution; excess refunded.
- Errors: Return empty data on invalid input (gas still consumed).

Listing 1: Solidity interface for BLS12-381 pairing precompile.

```
interface IBLS12381 {
    // Verify pairing equation: e(a1,b1) * e(a2,b2) * ... == 1
    // Input: pairs of (G1, G2) points concatenated
    // Output: bool (32 bytes, 0x01 or 0x00)
    function pairing(bytes calldata input)
        external view returns (bool);
}
```

2.3 Execution Model

Precompile functions execute in native Go, outside the EVM interpreter. The execution flow is:

Algorithm 1 Precompile Dispatch

Require: Call to address A with input I and gas G

Look up precompile handler P for address A
$g \leftarrow P.\mathsf{RequiredGas}(I)$ ▷ Compute required gas
if $G < g$ **then**
    return out-of-gas error
end if
$(R, \text{err}) \leftarrow P.\mathsf{Run}(I)$ ▷ Execute native code
if err $\neq$ nil **then**
    return empty data, consume all gas
end if
return R, refund $G - g$

3 BLS12-381 Precompiles

3.1 Curve Parameters

BLS12-381 [5] is a pairing-friendly elliptic curve with:

$$\mathbb{G}_1 : y^2 = x^3 + 4 \quad \text{over } \mathbb{F}_p, \quad p \approx 2^{381} \tag{1}$$

$$\mathbb{G}_2 : y^2 = x^3 + 4(u+1) \quad \text{over } \mathbb{F}_{p^2} \tag{2}$$

$$|\mathbb{G}_1| = |\mathbb{G}_2| = r \approx 2^{255} \tag{3}$$

Security level: approximately 128 bits. Embedding degree: 12.

3.2 Operations and Gas Costs

Table 2: BLS12-381 precompile gas costs vs. EIP-2537 proposal.

Operation	**Lux Gas**	**EIP-2537 Gas**	**Benchmark (ms)**
G1 Add	500	500	0.01
G1 Scalar Mul	12,000	12,000	0.37
G1 MSM (n points)	$12{,}000 \cdot n \cdot d(n)$	same	$0.37n \cdot d(n)$
G2 Add	800	800	0.02
G2 Scalar Mul	45,000	45,000	1.40
Pairing (k pairs)	$43{,}000k + 65{,}000$	$43{,}000k + 65{,}000$	$0.5k + 1.2$
Map to G1	5,500	5,500	0.16
Map to G2	110,000	110,000	3.30

The discount function $d(n)$ for multi-scalar multiplication (MSM) is:

$$d(n) = \max\left(0.4, \frac{1}{\lfloor \log_2 n \rfloor + 1}\right) \tag{4}$$

reflecting Pippenger's algorithm sublinear scaling.

3.3 Input Validation

Algorithm 2 BLS12-381 Pairing Input Validation

Require: Input bytes I

if $|I| = 0$ or $|I| \mod 288 \neq 0$ **then**
  return error (invalid length; each pair = 128 + 160 bytes)
end if
$k \leftarrow |I|/288$
for $j = 0$ to $k - 1$ **do**
  Parse $P_j \in \mathbb{G}_1$ from $I[288j : 288j + 128]$
  Parse $Q_j \in \mathbb{G}_2$ from $I[288j + 128 : 288(j + 1)]$
  Verify P_j is on curve $E(\mathbb{F}_p)$ and in subgroup $\mathbb{G}_1$
  Verify Q_j is on curve $E'(\mathbb{F}_{p^2})$ and in subgroup $\mathbb{G}_2$
  if any check fails **then**
    return error (point not on curve or not in subgroup)
  end if
end for
return valid

Subgroup checks are essential for security: accepting points outside the prime-order subgroup enables small-subgroup attacks that break pairing-based protocols [7].

3.4 Application: Groth16 Verification

BLS12-381 pairings enable on-chain Groth16 zk-SNARK verification:

Proposition 1 (Groth16 Verification Cost)**.** *Verifying a Groth16 proof with ℓ public inputs costs:*

$$G_{groth16} = (\ell + 1) \cdot 12{,}000 + 3 \cdot 43{,}000 + 65{,}000 = 12{,}000\ell + 206{,}000 \tag{5}$$

For $\ell = 10$ public inputs: 326,000 *gas, approximately \$0.02 at typical gas prices.*

4 BN254 Extended Precompiles

4.1 Motivation

While EIP-196/197 provide basic BN254 operations, they lack batch pairing verification and optimized MSM. Lux extends BN254 support with:

Table 3: BN254 extended precompile operations.

Operation	Lux Gas	EIP-196/197 Gas	Improvement
Pairing (2 pairs)	80,000	180,000	2.25x
Pairing (4 pairs)	120,000	340,000	2.83x
Batch pairing (random)	$40{,}000k + 40{,}000$	N/A	New
G1 MSM (n points)	$6{,}000n \cdot d(n)$	$6{,}000n$	$1/d(n)$x

4.2 Batch Pairing Verification

For verifying multiple independent pairing equations (e.g., batch Groth16 proofs), we use random linear combination:

Algorithm 3 Batch Pairing Verification

Require: k pairing equations $\{e(A_j, B_j) = e(C_j, D_j)\}_{j=1}^{k}$

Sample random $r_1, \ldots, r_k \xleftarrow{\$} \{1, \ldots, 2^{128}\}$
Compute $L = \prod_{j=1}^{k} e(r_j \cdot A_j, B_j)$
Compute $R = \prod_{j=1}^{k} e(r_j \cdot C_j, D_j)$
if $L = R$ **then**
    return ACCEPT
else
    return REJECT
end if

Theorem 2 (Batch Verification Soundness)**.** *If any individual pairing equation is false, the batch check rejects with probability* $\geq 1 - 2^{-128}$.

Proof. By the Schwartz-Zippel lemma, a non-zero polynomial of degree at most 1 in each r_j has at most $k \cdot 2^{-128}$ roots over the field, giving soundness error $\leq k/2^{128} \approx 2^{-128}$ for practical k. □

5 Secp256k1 Batch Verification

5.1 Design

The standard `ecrecover` precompile verifies one ECDSA signature per call at 3,000 gas. For applications verifying many signatures (e.g., multisig wallets, state channels, rollup batches), we provide batch verification:

Algorithm 4 Secp256k1 Batch Signature Verification

Require: n tuples $(h_j, r_j, s_j, \mathsf{pk}_j)$

Sample random weights $\alpha_1, \ldots, \alpha_n \xleftarrow{\$} \{1, \ldots, 2^{128}\}$
Compute $R_{\text{batch}} \leftarrow \sum_{j=1}^{n} \alpha_j \cdot R_j$ ▷ R_j from (r_j, s_j)
Compute $S_{\text{batch}} \leftarrow \sum_{j=1}^{n} \alpha_j \cdot s_j^{-1} \cdot h_j \cdot G + \sum_{j=1}^{n} \alpha_j \cdot s_j^{-1} \cdot r_j \cdot \mathsf{pk}_j$
if $R_{\text{batch}} = S_{\text{batch}}$ **then**
    return ACCEPT ALL
else
    Fall back to individual verification to identify invalid signatures
    return bitmap of valid signatures
end if

5.2 Gas Cost Model

$$G_{\text{batch}}(n) = 3{,}000 + 1{,}800 \cdot n \tag{6}$$

This is cheaper than n individual `ecrecover` calls ($3{,}000n$) for $n \geq 2$:

$$\text{Savings} = 1 - \frac{3{,}000 + 1{,}800n}{3{,}000n} = 1 - \frac{1}{n} - 0.6 = 0.4 - \frac{1}{n} \tag{7}$$

For $n = 100$: savings of 39% (180,000 vs. 300,000 gas).

Table 4: Batch vs. sequential secp256k1 verification costs.

Signatures	Sequential Gas	Batch Gas	Savings
10	30,000	21,000	30%
50	150,000	93,000	38%
100	300,000	183,000	39%
500	1,500,000	903,000	40%

6 Post-Quantum Signature Precompiles

6.1 Motivation

NIST's post-quantum cryptography standardization [6] has finalized ML-DSA (FIPS 204, formerly CRYSTALS-Dilithium) (ML-DSA) and FALCON (FN-DSA) as standard lattice-based signature schemes. Lux Network provides on-chain verification precompiles for both, enabling quantum-resistant smart contract authentication ahead of the quantum threat.

6.2 ML-DSA Verification

Definition 1 (Dilithium-3 Parameters)**.**

$$\textit{Security level: NIST Level 3 } (\approx 192\textit{-bit classical}) \tag{8}$$

$$\textit{Public key size: } 1{,}952 \textit{ bytes} \tag{9}$$

$$\textit{Signature size: } 3{,}293 \textit{ bytes} \tag{10}$$

$$\textit{Ring dimension: } n = 256, \quad q = 8{,}380{,}417 \tag{11}$$

Algorithm 5 Dilithium-3 Verification Precompile

Require: Input: pk (1,952 B) $\|$ σ (3,293 B) $\|$ m (variable)

1: Parse $\mathsf{pk} = (\rho, t_1)$ ▷ Seed and compressed key
2: Parse $\sigma = (\tilde{c}, z, h)$ ▷ Challenge, response, hint
3: Expand A from ρ using SHAKE-128
4: Verify $\|z\|_\infty < \gamma_1 - \beta$
5: Compute $w_1' = Az - ct_1 \cdot 2^d$
6: Use hint h to recover high bits
7: Recompute challenge $\tilde{c}' = H(\rho \| w_1' \| m)$
8: **if** $\tilde{c}' = \tilde{c}$ and $\|z\|$ norm check passes **then**
9: **return** 0x01 (valid)
10: **else**
11: **return** 0x00 (invalid)
12: **end if**

Gas cost: 35,000 (independent of message length for messages $<$ 1 KB; add 30 gas per additional 32-byte word).

6.3 FALCON Verification

Definition 2 (FALCON-512 Parameters)**.**

$$\textit{Security level: NIST Level 1 } (\approx 128\textit{-bit classical)} \tag{12}$$

$$\textit{Public key size: } 897 \textit{ bytes} \tag{13}$$

$$\textit{Signature size: } 666 \textit{ bytes (average)} \tag{14}$$

$$\textit{Ring dimension: } n = 512, \quad q = 12{,}289 \tag{15}$$

FALCON's advantage over ML-DSA is its smaller signature size (666 vs. 3,293 bytes), which reduces calldata costs:

Table 5: Post-quantum signature precompile comparison.

Scheme	PK (B)	Sig (B)	Verify Gas	Verify Time (ms)
Dilithium-2	1,312	2,420	28,000	0.22
Dilithium-3	1,952	3,293	35,000	0.30
Dilithium-5	2,592	4,595	48,000	0.43
FALCON-512	897	666	28,000	0.25
FALCON-1024	1,793	1,280	42,000	0.38

6.4 Calldata Cost Analysis

On-chain verification of post-quantum signatures requires transmitting large keys and signatures as calldata:

Table 6: Total on-chain cost including calldata (16 gas/byte non-zero).

Scheme	Verify Gas	Calldata Gas	Total Gas	vs. ECDSA
ECDSA (secp256k1)	3,000	1,088	4,088	1.0x
Dilithium-3	35,000	83,920	118,920	29.1x
FALCON-512	28,000	24,992	52,992	13.0x
FALCON-1024	42,000	49,168	91,168	22.3x

While post-quantum verification is 13–29x more expensive than ECDSA, the absolute cost remains practical ($< \$0.10$ per verification at typical gas prices), making it feasible for high-value transactions requiring quantum resistance.

6.5 Hybrid Authentication

We recommend a hybrid approach where transactions are signed with both ECDSA (for backward compatibility) and a post-quantum scheme (for forward security):

Definition 3 (Hybrid Signature)**.**

$$\sigma_{hybrid} = (\sigma_{ECDSA}, \sigma_{PQ}) \tag{16}$$

Verification succeeds iff both components verify. This provides security against both classical and quantum adversaries without requiring a hard fork to switch signature schemes.

7 Warp Message Verification Precompile

7.1 Interface

The Warp precompile verifies cross-appchain BLS aggregate signature proofs in a single call:

Listing 2: Warp verification precompile interface.

```
interface IWarpMessenger {
    struct WarpMessage {
        bytes32 sourceChainID;
        bytes32 destinationChainID;
        uint256 nonce;
        bytes payload;
        uint256 expiry;
        uint256 gasLimit;
    }

    function verifyMessage(
        WarpMessage calldata message,
        bytes calldata signature,   // 48 bytes BLS agg sig
        bytes calldata signerBitmap  // ceil(n/8) bytes
    ) external view returns (bool valid);
}
```

7.2 Gas Cost

The Warp precompile has a flat gas cost of 50,000 regardless of the source appchain's validator count, because all cryptographic work (public key aggregation and pairing check) is performed in optimized native code.

Table 7: Warp verification gas: precompile vs. pure Solidity.

Validators	Solidity (BLS ops)	Precompile
100	166,065	50,000
500	431,530	50,000
1,000	731,530	50,000

8 Gas Pricing Methodology

8.1 Calibration Principle

Gas prices are calibrated so that 1 gas $\approx$ 1 nanosecond of execution on reference hardware (AMD EPYC 7763, 2.45 GHz, single core). With Lux's 1 B gas block limit, this yields a target block execution time of $\sim$1 s on a single core—well within the 10 ms GPU BLS finality window when executed on GPU or multi-core C++ EVM (which processes 1 B gas in $\sim$48 ms). This ensures consistent block execution times regardless of transaction mix.

Definition 4 (Gas Price Invariant)**.** *For precompile P with execution time t_P milliseconds:*

$$G_P = \lfloor t_P \cdot 10^6 \cdot \kappa \rfloor \tag{17}$$

where $\kappa = 1.5$ is a safety factor accounting for variance across hardware.

8.2 Benchmark Results

Table 8: Precompile benchmarks on AMD EPYC 7763 (single core).

Precompile	Time (ms)	Calibrated Gas	Assigned Gas
BLS12-381 G1 Add	0.003	450	500
BLS12-381 G1 Mul	0.25	10,625	12,000
BLS12-381 Pairing (1 pair)	1.70	106,250	113,000
BLS12-381 Map to G1	0.05	5,000	5,500
BN254 Pairing (2 pairs)	1.10	73,750	80,000
secp256k1 Batch (100)	12.0	170,000	183,000
Dilithium-3 Verify	0.30	32,500	35,000
FALCON-512 Verify	0.25	26,000	28,000
Warp Verify	0.40	45,000	50,000

The ratio of assigned gas to calibrated gas ranges from 1.07 to 1.33, within the target $\kappa = 1.5$ safety factor.

8.3 DoS Resistance

Proposition 3 (Bounded Execution)**.** *All precompiles execute in $O(n)$ time where n is the input size in bytes, with constant factors bounded by $c \leq 50$ ns/byte. No precompile has super-linear time complexity in its input.*

Proof. MSM operations have time $O(n \log n / \log \log n)$ via Pippenger's algorithm, but input size for k points is $O(k)$ and execution is $O(k \log k)$. Since k is bounded by the block gas limit ($k \leq G_{\text{block}}/G_{\text{point}}$, where $G_{\text{block}} = 1{,}000{,}000{,}000$ on Lux), the absolute time is bounded. The 1 B gas limit enables ∼83,000 G1 scalar multiplications or ∼8,800 BLS12-381 pairing checks per block. □

9 Security Analysis

9.1 Input Validation

Every precompile validates inputs before processing:

1. Length checks: reject inputs that don't match expected format.
2. Point-on-curve checks: verify elliptic curve points lie on the correct curve.
3. Subgroup checks: verify points are in the prime-order subgroup.
4. Range checks: verify scalar values are in $[0, p)$ or $[0, r)$ as appropriate.

9.2 Side-Channel Resistance

Native precompile code uses constant-time implementations for all operations involving secret data:

- BLS signing (in the validator client, not the precompile) uses constant-time scalar multiplication.
- Pairing computations use constant-time Miller loop implementations.
- ECDSA batch verification uses constant-time modular inversion.

9.3 Known Attack Mitigations

Table 9: Attack vectors and mitigations.

Attack	Vector	Mitigation
Small subgroup	Points outside $\mathbb{G}_1$, $\mathbb{G}_2$	Mandatory subgroup check
Invalid curve	Points on twist	Point-on-curve validation
Rogue key (BLS)	Malicious key choice	Proof-of-possession requirement
Hash collision	Weak hash-to-curve	RFC 9380 compliant H2C
DoS via large input	Expensive MSM/pairing	Gas proportional to input size

10 Hash Function Precompiles

10.1 BLAKE2b-256

In addition to the standard Keccak-256 and SHA-256, Lux provides BLAKE2b-256 as a precompile for applications requiring BLAKE2b compatibility (e.g., Zcash interoperability, Substrate chains):

Table 10: Hash function precompile comparison.

Hash Function	Output (B)	Gas (64 B input)	Speed (ns/byte)
Keccak-256 (native)	32	30 + 6/word	3.2
SHA-256 (EIP-2)	32	60 + 12/word	2.8
BLAKE2b-256 (Lux)	32	25 + 5/word	1.9
BLAKE2f (EIP-152)	Variable	1/round	1.5

BLAKE2b is approximately 40% faster than Keccak-256 for the same input size, reflected in lower gas costs.

10.2 Poseidon Hash

For zero-knowledge applications, we provide a Poseidon hash precompile that is dramatically cheaper than computing Poseidon in Solidity:

Table 11: Poseidon precompile vs. Solidity implementation.

Operation	Solidity Gas	Precompile Gas
Poseidon hash (2 inputs)	45,000	3,000
Poseidon hash (4 inputs)	85,000	5,000
Poseidon hash (8 inputs)	165,000	9,000

This 15x gas reduction makes on-chain Merkle tree operations for privacy protocols and ZK rollups significantly more practical.

11 Precompile Upgrade Process

11.1 Adding New Precompiles

New precompiles are added through the Lux Improvement Proposal (LIP) process:

1. **LIP Draft**: Specification of interface, gas model, and security analysis.
2. **Reference Implementation**: Go implementation in the Lux VM codebase.
3. **Audit**: External security audit for cryptographic correctness.
4. **Testnet Deployment**: 30-day testing period on Lux testnet.
5. **Governance Vote**: Supermajority validator approval.
6. **Mainnet Activation**: Scheduled activation at a future block height.

11.2 Appchain-Specific Precompiles

Individual appchains can enable additional precompiles not available on the primary network:

Definition 5 (Custom Appchain Precompile)**.** *A appchain creator can register custom precompiles at addresses `0x03...01` through `0x03...FF` in their appchain's VM configuration. These precompiles run only on that appchain and do not affect other chains.*

Example use cases:

- Gaming appchains: Verifiable random function (VRF) precompile for provably fair randomness.
- DeFi appchains: Fixed-point arithmetic precompile for precise financial calculations.
- AI appchains: Tensor operation precompile for on-chain ML inference verification.

11.3 Deprecation Policy

Precompiles are never removed (backward compatibility) but can be deprecated:

1. Gas cost increased to discourage new usage.
2. Warning emitted in node logs.
3. Replacement precompile documented.

12 Comparison with Ethereum Precompiles

Table 12: Lux vs. Ethereum precompile coverage.

Capability	Ethereum	Lux	Status
secp256k1 recovery	ecrecover	ecrecover + batch	Extended
SHA-256, RIPEMD-160	EIP-2, EIP-3	Same + BLAKE2b	Extended
BN254 operations	EIP-196/197	Same + batch pairing	Extended
BLS12-381 operations	EIP-2537 (pending)	Deployed	Ahead
Post-quantum sigs	Not proposed	ML-DSA + FALCON	Unique
Cross-chain verify	Not applicable	Warp precompile	Unique
Modular exp	EIP-198	Same	Equivalent

Lux's precompile suite is a strict superset of Ethereum's current and proposed precompiles, with additional capabilities for post-quantum cryptography and cross-chain verification.

13 Related Work

EIP-2537 [4] proposes BLS12-381 precompiles for Ethereum with gas costs derived from benchmarking on commodity hardware. Our BLS12-381 implementation follows the same specification but benefits from tighter integration with the Lux VM.

Post-quantum cryptography on blockchains has been explored by Chen et al. [13], who analyze the feasibility of lattice-based signatures for blockchain authentication. NIST's standardization process [6] finalized ML-DSA (FIPS 204, formerly CRYSTALS-Dilithium) and FALCON in 2024.

Batch verification techniques were introduced by Bellare et al. [11] and adapted for blockchain by Boneh et al. [9]. The Schwartz-Zippel soundness analysis follows standard techniques from probabilistic verification [12].

14 Conclusion

Lux Network's custom EVM precompiles provide native-speed cryptographic operations that enable advanced smart contract applications:

1. Full BLS12-381 support enables on-chain zk-SNARK verification and aggregate signature checking.
2. Batch secp256k1 verification achieves 40% gas savings for multi-signature operations.
3. Post-quantum precompiles (ML-DSA, FALCON) provide quantum-resistant authentication at practical gas costs.
4. The Warp verification precompile enables constant-gas cross-appchain message validation.
5. Gas pricing calibrated to execution time within a 1.5x safety factor ensures predictable block execution.

The precompile suite positions Lux Network as the most cryptographically capable EVM platform, supporting both current security needs and future quantum resistance requirements.

14.1 Testing and Formal Verification

Each precompile is validated through a multi-layer testing strategy:

1. **Unit tests**: Known-answer tests (KATs) from IETF and NIST specifications for every supported curve and hash function. Coverage includes edge cases: identity elements, points at infinity, maximum-length inputs, and malformed encodings.
2. **Differential testing**: Outputs are compared against reference implementations (BLST for BLS12-381, Gnark for BN254, PQClean for ML-DSA/FALCON) across 10^6 random inputs per operation.
3. **Fuzzing**: Continuous fuzzing with AFL++ targeting input parsing, gas calculation, and memory allocation paths. Each precompile undergoes 10^9 fuzzer iterations before mainnet activation.
4. **Gas invariant checking**: Automated verification that gas charged is always $\geq$ measured execution time $\times$ gas-per-nanosecond rate, across all benchmark hardware configurations.

All precompile implementations are open-source and undergo continuous integration testing against the reference test vectors maintained in the `lux-precompile-tests` repository. New test vectors are added for each EVM upgrade cycle.

The testing infrastructure includes a dedicated precompile benchmarking harness that measures execution time on reference hardware (AWS c6g.xlarge ARM instances) and automatically updates gas tables when performance characteristics change due to compiler or library upgrades. The benchmarking results are published in each network upgrade's release notes, providing transparency into gas pricing decisions and enabling third-party auditors to independently verify the gas cost model.

14.2 Precompile Address Allocation

The Lux EVM reserves address range `0x0300`–`0x03FF` for network-specific precompiles, avoiding conflicts with the Ethereum-standard range (`0x01`–`0x0A`) and the proposed EIP-2537 BLS range (`0x0B`–`0x0F`). The allocation scheme groups precompiles by cryptographic family: `0x0300`–`0x030F` for BLS12-381 operations, `0x0310`–`0x031F` for BN254, `0x0320`–`0x032F` for post-quantum signatures, `0x0330`–`0x033F` for hash functions, and `0x0340`–`0x034F` for Warp messaging verification. This structured allocation simplifies tooling, auditing, and documentation.

References

[1] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," 2014.

[2] C. Reitwiessner, "EIP-196: Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128," 2017.

[3] C. Reitwiessner and V. Buterin, "EIP-197: Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128," 2017.

[4] A. Vlasov, "EIP-2537: Precompile for BLS12-381 curve operations," 2020.

[5] S. Bowe, "BLS12-381: New zk-SNARK elliptic curve construction," *Zcash Blog*, 2017.

[6] NIST, "Post-quantum cryptography standardization: Selected algorithms," 2024.

[7] P. S. L. M. Barreto and M. Naehrig, "Pairing-friendly elliptic curves of prime order," in *SAC*, 2006.

[8] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the Weil pairing," in *ASIACRYPT*, 2001.

[9] D. Boneh, M. Drijvers, and G. Neven, "Compact multi-signatures for smaller blockchains," in *ASIACRYPT*, 2018.

[10] J. Groth, "On the size of pairing-based non-interactive arguments," in *EUROCRYPT*, 2016.

[11] M. Bellare, J. A. Garay, and T. Rabin, "Fast batch verification for modular exponentiation and digital signatures," in *EUROCRYPT*, 1998.

[12] J. T. Schwartz, "Fast probabilistic algorithms for verification of polynomial identities," *JACM*, 1980.

[13] L. Chen et al., "Post-quantum cryptography for blockchain: A survey," *ACM Computing Surveys*, 2021.

[14] L. Ducas et al., "ML-DSA (FIPS 204, formerly CRYSTALS-Dilithium): Algorithm specifications and supporting documentation," *NIST PQC Round 3*, 2022.

[15] T. Prest et al., "FALCON: Fast-Fourier lattice-based compact signatures over NTRU," *NIST PQC Round 3*, 2020.

[16] N. Pippenger, "On the evaluation of powers and related problems," in *FOCS*, 1976.

[17] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[18] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Yellow Paper*, 2014.

[19] S. D. Galbraith, "Mathematics of public key cryptography," Cambridge University Press, 2012.

[20] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *FOCS*, 1994.

[21] M. Mosca, "Cybersecurity in an era with quantum computers," *IEEE Security & Privacy*, 2018.

[22] D. J. Bernstein and T. Lange, "Post-quantum cryptography," *Nature*, vol. 549, pp. 188–194, 2017.

[23] A. Faz-Hernandez et al., "Hashing to elliptic curves," *RFC 9380*, 2023.

[24] C. Costello, T. Lange, and M. Naehrig, "Faster pairing computations on curves with high-degree twists," in *PKC*, 2010.

[25] S. Boneh et al., "BLS signature scheme," *IETF Internet-Draft*, 2022.

[26] V. Lyubashevsky, "Lattice signatures without trapdoors," in *EUROCRYPT*, 2012.