Signed Software, Reproducible Builds, and Quantum-Resilient Release Pipelines

Zach Kelling

Lux Industries

2025

Abstract

A blockchain node binary is the ultimate trust anchor: if the binary is compromised, no amount of cryptographic protocol design matters. We describe Lux Network's release pipeline, which enforces deterministic builds, ML-DSA (FIPS 204) signed binaries, and an attestation chain from source commit to deployed container image. Every release artifact carries a hybrid BLS+ML-DSA signature over a content-addressed manifest that includes source commit hash, build environment hash, compiler version, and dependency lockfile digest. Reproducibility is verified by independent rebuilders, and attestations are published to a transparency log anchored in Q-Chain. The pipeline covers `luxd`, EVM plugins, and subnet VM binaries across Linux (amd64, arm64) targets.

1 Introduction

Supply chain attacks on software dependencies have escalated from theoretical concern to operational reality. The SolarWinds incident (2020), the xz/liblzma backdoor (2024), and numerous npm/PyPI compromises demonstrate that build pipelines are high-value targets. For blockchain infrastructure, a compromised node binary can steal funds, halt consensus, or silently corrupt state.

Lux Network addresses this with three interlocking guarantees:

1. **Deterministic builds**: Given the same source commit and build environment, any party produces the identical binary.
2. **Quantum-resilient signatures**: Release artifacts are signed with ML-DSA-65 (FIPS 204) in addition to classical Ed25519, ensuring signature validity even against future quantum adversaries.
3. **Attestation transparency**: Every step from source to deployment is recorded in an append-only log anchored in Q-Chain, enabling public audit.

1.1 Scope

The pipeline covers:

- `luxd` node binary (Go, `CGO_ENABLED=0`)
- EVM plugin binaries (Go, compiled against `luxfi/evm`)
- Subnet VM binaries (application-specific)
- Container images (`ghcr.io/luxfi/*`, `--platform linux/amd64`)

2 Deterministic Build System

2.1 Build Environment Specification

Each release specifies a locked build environment:

```
build-env.lock:
  go_version: 1.22.4
  os: linux
  arch: amd64
```

```
cgo: disabled
ldflags: -s -w -X main.version=v1.23.23
go_sum_hash: sha256:<digest>
toolchain_hash: sha256:<digest>
```

The build environment itself is a content-addressed container image. The image is built once, signed, and reused for all builds at that Go version. Changing the Go version requires a new environment image with a new signature.

2.2 Reproducibility Protocol

function BUILD(commit, envImage)
$source \leftarrow$ GitCheckout($commit$)
$deps \leftarrow$ GoModDownload($source$)
$binary \leftarrow$ GoBuild($source, deps, envImage$)
$digest \leftarrow$ SHA256($binary$)
return $(binary, digest)$
end function

Two properties ensure reproducibility:

1. **No embedded timestamps**: Build flags strip debug info (`-s -w`) and inject only the version string and commit hash. No build time, hostname, or path is embedded.

2. **Locked dependencies**: `go.sum` is committed and verified. The dependency fetch is hermetic—no network access during compilation.

2.3 Independent Rebuilder Verification

At least two independent rebuilders (community-operated, not Lux Industries) reproduce each release. A release is considered verified when ≥ 2 independent rebuilders produce the identical binary digest.

Artifact	Target	Binary Size
luxd	linux/amd64	∼48 MB
luxd	linux/arm64	∼45 MB
evm-plugin	linux/amd64	∼32 MB

Table 1: Typical artifact sizes for Lux v1.23.x releases.

3 Hybrid Signature Scheme

3.1 Release Manifest

Each release produces a signed manifest:

```
manifest.json:
{
  "version": "v1.23.23",
```

```
  "commit": "abc123def456",
  "artifacts": [
    {
      "name": "luxd-linux-amd64",
      "sha256": "<digest>",
      "size": 50331648
    }
  ],
  "build_env_hash": "<digest>",
  "go_version": "1.22.4",
  "go_sum_hash": "<digest>",
  "timestamp": "2025-03-01T00:00:00Z"
}
```

3.2 Dual Signature

The manifest is signed with both Ed25519 (classical) and ML-DSA-65 (post-quantum):

$$\sigma_{\text{ed25519}} = \text{Sign}_{\text{Ed25519}}(sk_c, H(\text{manifest})) \tag{1}$$

$$\sigma_{\text{mldsa}} = \text{Sign}_{\text{MLDSA}}(sk_{pq}, H(\text{manifest})) \tag{2}$$

$$\text{release} = (\text{manifest}, \sigma_{\text{ed25519}}, \sigma_{\text{mldsa}}) \tag{3}$$

The signing keys are held in an HSM [1]. The ML-DSA key is generated and stored in the same HSM boundary as the Ed25519 key. Neither key is extractable.

3.3 Verification

Validators verify release signatures before applying upgrades:

1: **function** VerifyRelease(release)
2: $h \leftarrow \text{SHA256}(release.\text{manifest})$
3: $v_c \leftarrow \text{VerifyEd25519}(pk_c, \sigma_{\text{ed25519}}, h)$
4: $v_{pq} \leftarrow \text{VerifyMLDSA}(pk_{pq}, \sigma_{\text{mldsa}}, h)$
5: **return** $v_c \wedge v_{pq}$
6: **end function**

4 Attestation Transparency

4.1 Attestation Chain

Every build step produces an attestation record:

4.2 Q-Chain Anchor

Attestation records are batched and their Merkle root is submitted to Q-Chain as a governance transaction. This provides a tamper-evident, publicly auditable record that cannot be rewritten without breaking Q-Chain consensus.

Validators can verify that the binary they are running corresponds to a release whose attestation root is recorded on-chain:

Step	Attestation	Signer
Source	Commit signed (GPG/SSH)	Developer
Build	Binary digest + env hash	CI runner
Rebuild	Matching digest confirmation	Independent rebuilder
Sign	Manifest + dual signature	Release HSM
Publish	Container image digest	Registry (GHCR)
Deploy	Running image digest	Validator node

Table 2: Attestation chain from source to deployment.

1: **function** VerifyDeployment(binaryDigest)
2: $root \leftarrow$ QChain.LatestAttestationRoot()
3: $proof \leftarrow$ FetchMerkleProof($binaryDigest, root$)
4: **return** VerifyMerkleProof($proof, root, binaryDigest$)
5: **end function**

5 Container Image Pipeline

Container images are built from the signed binary, not from source. The Dockerfile is minimal:

```
FROM debian:bookworm-slim
COPY --from=signed luxd /usr/local/bin/luxd
COPY --from=signed plugins/ /usr/local/lib/luxd/plugins/
ENTRYPOINT ["luxd"]
```

Images are built using Kaniko (no Docker daemon) inside the Kubernetes cluster, pushed to `ghcr.io/luxfi/node:<tag>`, and signed with Cosign using the same ML-DSA key. The image digest is included in the attestation chain.

Images are built for `linux/amd64` only. Multi-arch images are produced by CI runners on the target architecture, not by cross-compilation or emulation.

6 Threat Model and Mitigations

Threat	Mitigation	Detection
Compromised CI runner	Reproducible builds	Rebuilder mismatch
Dependency backdoor	Locked go.sum, hermetic build	Hash mismatch on rebuild
Signing key theft	HSM-bound keys, dual signature	Requires both key types
Quantum signature forge	ML-DSA alongside Ed25519	Hybrid check fails
Binary swap post-sign	Content-addressed manifest	Digest verification

Table 3: Threat mitigations in the release pipeline.

7 Conclusion

The distance from source code to running validator is the most security-critical path in a blockchain system. Lux's release pipeline closes every link in this chain: deterministic builds eliminate the "it

built differently" class of attacks, hybrid ML-DSA+Ed25519 signatures protect against both current and future adversaries, and Q-Chain-anchored attestation transparency makes every step publicly auditable. Independent rebuilders provide the final check: if the binary cannot be reproduced, it is not released.

References

[1] Kelling, Z. (2025). *Secure Enclave, HSM, and Threshold Boundary Design*. Lux Industries Research.

[2] Kelling, Z. (2025). *Quasar: Quantum-Secure Multi-Engine Consensus with Triple-Proof Quantum Finality*. Lux Industries Research.

[3] NIST (2024). *FIPS 204: Module-Lattice-Based Digital Signature Standard (ML-DSA)*. National Institute of Standards and Technology.

[4] Reproducible Builds Project (2024). *Reproducible Builds: A Set of Software Development Practices*. `https://reproducible-builds.org/`.

[5] Sigstore Project (2024). *Sigstore: A New Standard for Signing, Verifying, and Protecting Software*. Linux Foundation.

[6] SLSA Contributors (2024). *Supply-chain Levels for Software Artifacts (SLSA)*. `https://slsa.dev/`.