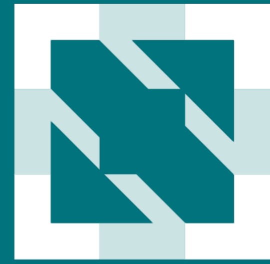


KubeCon | CloudNativeCon

India 2026

About US

Swastik Gour

Assoc. Engineer-Product @ Improving

Sharvil Bhatt

Ex-Engineer @ Reliance Industries Limited

Introduction and the Problem Space

The tension between governance and vendor IP protection

KubeCon | CloudNativeCon
India 2026

What is an SBOM

A Software Bill of Materials is an inventory of every component, library, and dependency that makes up a piece of software.

- Required by NIST SSDF, US Executive Order 14028, EU Cyber Resilience Act
- Standard formats: CycloneDX, SPDX
- Typical enterprise SBOM contains 1,000 to 50,000 components

Two demands that conflict

Customers and regulators want

- Full transparency
- Ability to verify
- Ongoing monitoring

Vendors resist publishing because

- SBOM is a vulnerability map
- Exposes architecture decisions
- Reveals supplier relationships

Traditional approaches force a binary choice — publish everything or hide everything.

The recipe analogy is misleading

An SBOM is not a recipe for software the way a recipe is for a dish.

- A culinary recipe gives 90% of what you need to reproduce the dish
- An SBOM gives 5 to 10% — components without architecture, business logic, configuration, or engineering judgment
- Software is hard to replicate even with a full SBOM in hand

Competitive replication is not the real risk.

What an exposed SBOM actually enables

The real threat is targeted attack intelligence, not replication.

- **Vulnerability mapping** — cross-reference SBOM against CVE databases for ready-made exploits
- **Internal architecture exposure** — proprietary library names reveal design decisions
- **Supply chain intelligence** — supplier relationships become attack vectors
- **Regulatory and licensing exposure** — undisclosed third-party components become legal risk

Three properties that must coexist

01

Verifiability

Anyone can confirm the SBOM has not been tampered with

02

Privacy

Component details remain confidential

03

Selective disclosure

Auditors can verify specific components without seeing the rest

These three are not contradictory if you build the right cryptographic structure.

Two roots, two purposes

PUBLIC_ROOT

32 bytes, shared with everyone, proves SBOM has not changed

COMMITMENT_ROOT

32 bytes, shared with auditor only, enables selective disclosure

Built from the same data, using the same primitive — SHA-256.

Layer 1: How the Internal Tree Works

Two hash functions. Both SHA-256. Separated by a 1-byte domain prefix:

```
H_leaf(data)        = SHA-256( 0x00 ‖ data )
H_node(left, right) = SHA-256( 0x01 ‖ left ‖ right )
```

Why two prefixes?

Without them, an attacker can substitute an internal node for a leaf and produce the same root — a second-preimage attack. The 0x00/0x01 byte makes the two hash spaces irreconcilable.

The root of this tree is the **PUBLIC_ROOT** — 32 bytes, shared with everyone. Any change to any component flips it. No component data is exposed.

- Tree structure: leaves are component hashes, internal nodes combine children pairwise up to root
- Odd number of leaves: last leaf is duplicated to keep binary structure
- No randomness — deterministic, so the same SBOM always produces the same PUBLIC_ROOT

KubeCon CloudNativeCon India 2026

Layer 2: The Commitment Scheme

Per component, 3 steps

Step 1 — Hash: `h_i = H_leaf( component_i )`

Step 2 — Nonce: `r_i ← {0,1}^256  (random)`

Step 3 — Commit: `C_i = SHA-256( h_i ‖ r_i )`

Build Merkle tree from {C_0 ... C_n} →

COMMITMENT_ROOT

Two formal properties

Hiding (computational): Given C_i, it is infeasible to learn h_i without r_i. Security rests on SHA-256 preimage resistance.

Binding (info-theoretic): Cannot find (h’, r’) with h’ ≠ h such that SHA-256(h’ ‖ r’) = C_i. Collision resistance gives 2^128 security.

Re-randomizable: Generate new nonces r’_i to refresh COMMITMENT_ROOT without touching h_i or PUBLIC_ROOT.

In plain English: each component is sealed in an envelope. The vendor can open one without revealing the others.

KubeCon | CloudNativeCon India 2026

Worked Example: Sealing 4 Components

Input: `openssl:3.1.3  nginx:1.25.2  postgres:15.2  redis:7.0.11`

Step 1 — Leaf hashes (internal tree, no randomness):

```
h_0 = SHA-256(0x00 || "openssl|3.1.3")  →  a3f2...    h_2 = SHA-256(0x00 || "postgres|15.2")  →  2b44...
h_1 = SHA-256(0x00 || "nginx|1.25.2")   →  7c91...    h_3 = SHA-256(0x00 || "redis|7.0.11")   →  f801...
```

`Merkle(h_0...h_3) →` **PUBLIC_ROOT = 9d3a...**

Step 2 — Random nonces (secret, never published):

```
r_0 = os.urandom(32) → 88c1...    r_1 = os.urandom(32) → 3f72...    r_2 = 5a10...    r_3 = 0c6e...
```

Step 3 — Commitments:

```
C_0 = SHA-256(h_0 || r_0) → 4d8f...    C_1 → 91b2...    C_2 → c73a...    C_3 → 0e5f...
```

`Merkle(C_0...C_3) →` **COMMITMENT_ROOT = 1f7b...** (sent to auditor only)

KubeCon CloudNativeCon India 2026

openssl Updated — Here’s the Proof

Vendor disclosure bundle

```
component: “openssl”
new_version: “3.1.4”
h_i: SHA-256(0x00 || “openssl|3.1.4”)
r_i: <nonce for slot 0>
proof_path: [sib_1, sib_23]
(O(log n) siblings)
```

nginx, postgres, redis: nonces never revealed. Still sealed.

Auditor verifies in 3 steps

1. Re-derive leaf hash:

```
   h_check = SHA-256(0x00 || “openssl|
3.1.4”)
   assert h_check == h_i ✓
```

2. Re-derive commitment:

```
   C_check = SHA-256( h_i || r_i )  ✓
```

3. Walk Merkle path to root:

```
   acc = C_check
   acc = SHA-256(0x01 || acc || sibling_1)
   assert acc == COMMITMENT_ROOT  ✓
```

Proof size: 481 bytes for 10,000 components (O(log n) siblings). Only 1 component revealed. 3 remain sealed.

Re-randomization: Same Data, New Nonces

The operation

```
r'_i ← os.urandom(32)   # new nonce
C'_i = SHA-256( h_i || r'_i )   # same h_i
Rebuild Merkle tree → new COMMITMENT_ROOT
```

Why? Breaks linkability across audit sessions. Old nonces can't be replayed to correlate disclosures over time.

What changes, what doesn’t

```
PUBLIC_ROOT          — unchanged ✓
COMMITMENT_ROOT      — rotated ↺
Component data       — unchanged ✓
Nonces r_0...r_n     — replaced ↺
```

Tested: 5 consecutive rounds. PUBLIC_ROOT never changed. Proofs valid every round.

Re-randomization cost: 1.1 ms for 1K components, 11 ms for 10K. Fast enough to run on any schedule.

Performance: Build Time and Proof Operations

Build time — O(n), runs once per update:

```
100 components:    0.26 ms    |  10,000 components:    21.6 ms
1,000 components:   2.23 ms   |  100,000 components:  247 ms
```

Linear scaling, ~2.5 μs per component. A 10K enterprise SBOM builds in 22 ms — negligible in CI/CD.

Proof ops — O(log n), the workhorse of production:

```
1K components:   gen 0.006 ms  |  verify 0.007 ms  |  size  384 bytes
10K components:  gen 0.009 ms  |  verify 0.011 ms  |  size  481 bytes
100K components: gen 0.042 ms  |  verify 0.027 ms  |  size  625 bytes
```

Sub-millisecond at every scale. 625 bytes for 100K components — fits in a single HTTP request.

KubeCon CloudNativeCon India 2026

Stress-Tested: 1,000 Auditors, 100,000 Components

Multi-auditor scaling

```
10 auditors:   594 ms  (baseline)
50 auditors:   612 ms  (+0.36 ms/auditor)
100 auditors:  602 ms  (+0.08 ms/auditor)
500 auditors:  649 ms  (+0.11 ms/auditor)
```

50× more auditors = <10% overhead. Near-linear scaling.

Security validation

- ✓ 50,000 proofs verified — 100% pass rate
- ✓ Tampered component data — rejected
- ✓ Tampered nonce — rejected
- ✓ Tampered proof path — rejected
- ✓ Selective disclosure: 72% reduction in data sent

2.6M disclosures/second throughput. 100% tamper detection. All security properties verified at 100K scale.

Two Channels, One Boundary

PUBLIC channel (everyone)

CI/CD → Public Registry

Publishes: PUBLIC_ROOT only (32 bytes)
Anyone can monitor changes, detect rollbacks, verify integrity. Zero component exposure.

Phase 1 (today): up to 100 auditors, 10K components, 22 ms/update

PRIVATE channel (auditor only)

CI/CD → Auditor Interface (mTLS)

Sends: COMMITMENT_ROOT + decommitment blobs for changed components only
Auditor verifies cryptographically. Unchanged components: nonces never sent, never leaked.

Phase 2 (incremental updates): 0.01 ms per update — 2,200× faster, $1/month at scale

Three actors, minimal trust required. No shared secrets between public and auditor channels.

Why Pure ZKP Doesn't Fit Real CI/CD Pipelines

Performance reality

- **8 seconds per proof** — CI/CD runs hundreds of builds per day; this blocks the pipeline
- **Circuit recompile on every schema change** — ~10s, and SBOM schemas change often
- **Fixed at 96 packages, 8 queries** — real enterprise SBOMs have 5,000+ components
- **No incremental updates** — adding one package forces full proof regeneration from scratch

Operational barriers

- **16 MB trusted setup + Powers of Tau ceremony** — complex infrastructure most platform teams cannot maintain
- **Poseidon hash ≠ SHA-256** — incompatible with Syft, Grype, and every existing SBOM tool
- **Circom + snarkjs + Rust toolchain** — not standard DevOps tooling; requires cryptography specialists
- **1-bit output = debugging blackout** — when non-compliant, you cannot see which query failed or why

ZKP gives mathematical perfection at the cost of operational reality. The commitment scheme delivers 2.6M disclosures/second with zero toolchain overhead.

The CI/CD Pipeline: 12 Steps, Real Tools

End-to-end flow (12 steps)

1. Docker build
2. cosign sign (image signing)
3. syft → CycloneDX SBOM
4. grype → vulnerability scan
5. in-toto attest (5 signed links)
6. merkleSBOM commitment → roots
7. TUF metadata signed + published
8. OPA policy gate
9. remote auditor verify
10. kind deploy (Kubernetes)
11. integration tests
12. NIST compliance report

Tool integration

Tool	Role	Standard
syft	SBOM generation	CycloneDX 1.6
grype	Vuln scan	CVE/NVD
cosign	Image signing	Sigstore
in-toto	Supply-chain attest	SLSA/in-toto
TUF	Root distribution	TUF spec
OPA	Policy enforcement	Rego
kind	Deployment target	Kubernetes
merkleSBOM	Privacy commitment	SHA-256 Merkle

The point is interoperability — the scheme plugs into the existing toolchain rather than replacing it.

in-toto: Attesting the Build

What it is

A CNCF supply-chain framework that cryptographically records who did what to an artifact, so the whole build can be verified end-to-end.

Each pipeline step produces a signed link file containing its inputs (materials) and outputs (products), hashed by SHA-256. The layout file enforces step ordering and expected artifacts.

How we use it

- Signed link metadata for 5 steps: build-image, generate-sbom, scan-vulns, create-commitment, policy-check
- Each link records materials → products by SHA-256 (e.g. SBOM → grype-results.json → merkle_roots.json)
- Supply-chain layout enforces step order and expected artifacts — a tampered or skipped step is detectable
- Satisfies NIST SSDF PW.1.1 (security-focused build environments)

SLSA compliance out of the box: every artifact transition in the pipeline is cryptographically witnessed.

TUF: Distributing the Roots Safely

What it is

The Update Framework, a CNCF project that protects how signed metadata reaches consumers — with built-in resistance to rollback and key-compromise attacks.

Without TUF, an attacker who compromises the distribution channel can serve stale or tampered roots. TUF makes that detectable by design.

How we use it

- Protects merkle_roots.json (PUBLIC_ROOT + COMMITMENT_ROOT) as a signed TUF target — recorded by length and SHA-256
- Four-role key hierarchy: root → targets → snapshot → timestamp, each independently signed
- Expiration-based freshness: 365-day root, 90-day targets, 30-day snapshot, 1-day timestamp — stale roots are rejected
- Satisfies NIST SSDF PS.2.1 (secure software distribution)

The Merkle root is only useful if consumers can trust they received the right one. TUF closes that gap without a PKI.

OPA: Gating on Policy (Integrity ≠ Security)

What it is

Open Policy Agent, a CNCF general-purpose policy engine using the Rego language. The layer that gates deployment.

Key distinction: the Merkle commitment proves the SBOM was not tampered with. OPA proves the SBOM is actually safe to ship.

How we use it

- Three Rego policies: SBOM completeness, vulnerability threshold (no Critical; High → warning), NIST compliance (all steps completed)
- OPA Gatekeeper extends into Kubernetes admission: pods must carry sbom.merkle/pipeline annotation and SBOM_PUBLIC_ROOT env var
- **Result:** a vulnerable image passes commitment verification but FAILS the OPA policy — demonstrating the two layers are independent

Integrity without policy is not security. A perfectly committed SBOM full of Log4Shell is still a blocked deployment.

Kubernetes (kind) + Sigstore/cosign: Runtime & Signing

What it is

Kubernetes is the CNCF container orchestrator; kind (Kubernetes in Docker) runs it locally for the demo. cosign (Sigstore/OpenSSF) handles image signing at every build.

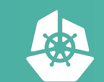
Together they represent the runtime and signing halves of the pipeline — where the commitment finally becomes an enforced constraint.

How we use it

- kind deploys the image with Merkle roots injected as a ConfigMap; multi-node pins secure/medium/legacy variants to tiered nodes via nodeSelector
- OPA Gatekeeper constraint enforces SBOM-commitment annotations at admission time — unsigned or unannotated pods are rejected
- cosign key pair + image-signing step completes provenance from build to deploy
- Full loop: signed image → attested build → committed SBOM → policy-gated, verifiable runtime

Every CNCF tool in this pipeline was already in your stack. The commitment scheme is the missing cryptographic link between them.

Github Repository: https://github.com/Sharvil2704/Commit-then-disclose.git



THANK YOU