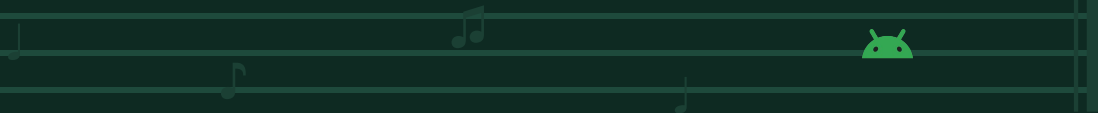


SOSTENUTO

Scaling & Sustaining Android Binary Transparency

Billy Lau | Android Security & Privacy, Google



Where we left off

Odyssey · 2023	Crescendo · 2025	Today
Pixel firmware: the first Android binary transparency log	Extended transparency toward the application layer	GPAT + Mainline Modules in production
Low-cadence, monolithic OS builds	Left open: every first-party app, the actual APKs, trustworthy automation	Continuous ingestion, run as a service: TLUaaS
One build, one hash, curated by hand		Witnessed checkpoints, now

Same primitive, same key ceremony. Different tempo, and a much larger ensemble.

Firmware logging vs. app logging

Dimension	Firmware (OS images)	Apps (GPAT / Mainline)
Release cadence	Monthly, scheduled	Continuous, always in flight
Artifacts per version	One image, one hash	Variant matrix: ABI x density x feature splits
Entry curation	Manual, human-gated	Automated, pipeline-driven
Source of truth	Build system, visible pre-prod	Prod-only: signing keys, manifests, distribution streams
Cost of a bad entry	Irreversible	Irreversible, at orders of magnitude more writes



What we log today



GPAT

Google Product Application Transparency

Standalone product apps, continuous release

`com.google.android.gms`

`com.google.android.youtube`



Mainline Modules

Modular OS components delivered through app update channels

OS semantics, app delivery mechanics

Same log design, different producer

https://developers.google.com/android/binary_transparency/google_apk/overview
https://developers.google.com/android/binary_transparency/mainline_modules/overview



How far up the stack

Verified boot anchors the bottom. Crescendo pushed the guarantee upward. This is where it reaches today.

Applications



Third-party apps

Outside this program



Google first-party apps · GPAT

Continuous, per artifact

Platform services



Google Play services · GMS Core

Continuous, per artifact · logged as part of GPAT · used by first- and third-party apps alike

Operating system



Pixel system images

Per build



Mainline modules

Continuous, per artifact

Boot



Pixel bootloader · verified boot / VBMeta

Per build

Same log design at every layer. What changes is the tempo - sometimes within one layer.

One score, many parts

1 -> N

A single logical version does not yield a single binary hash



ABI



Screen density



Dynamic feature splits

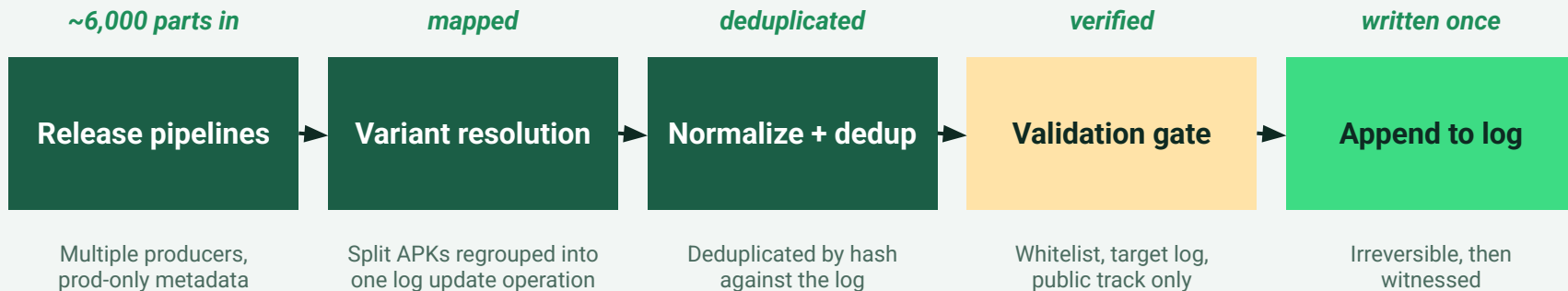


Distribution stream

Every part has to be mapped, deduplicated and verified before one irreversible write



TLUaaS: Transparency Log Updater as a Service



Append-only transparency log -> checkpoints cosigned by external witnesses



Devices and auditors verify inclusion proofs (Uraniborg, ecosystem verifiers)

Same song, new key

The log design did not change. The machinery underneath it did - and the replacement was already running.

What we outgrew

One file listed every entry in the log, and every publish rewrote it from entry zero.

At roughly 1.8 million entries it reached the 200 MB single-file ceiling of our static serving layer. That is where the design ran out.

The ceiling was in how we published, not in what we logged. 2026/01 is closed and complete at 1,797,152 entries - still published, still verifiable.

What we moved to: Tessera

Tiled logs: 256 entries to a tile, static and cacheable. Publishing appends a tile instead of rewriting a monolith.

The old layout produced a tile per entry, and we could not change that without breaking every existing client.

Batching and witness cosigning come with the library.

A format change is a rewrite - the one thing we promised never to do. So we froze the old shard and opened a new one:

google1p/apk/2026/01	legacy	archived read-only, complete at 1,797,152 entries
google1p/apk/2026/02	tessera	tiled, Ed25519, current

Tessera is open source, from transparency.dev, and the successor to Trillian. We run the upstream library with our own storage driver - not a fork.

Variant resolution: reassembling the score

Start from the release, not the file

We walk everything built for a release - not whatever files happen to show up.

Walk every variant

Each separately built variant of that release, public tracks only.

Collect every part

The base APK, every split, and the exact file delivered - each one hashed.

Then only what is new goes forward: anything already logged is skipped.

Why this is the hard part

Miss a part, and a device runs something the log cannot vouch for.

Mislabel a part, and the log vouches for a release that never existed.

Commit discipline: you cannot un-play a note

BEFORE THE LINE

Reversible

Resolve, collect, check, retry.
Throw it away and start again.

Mistakes here are cheap.



AFTER THE LINE

Permanent

Public, and quotable by anyone -
including someone arguing we shipped
something we didn't.

Mistakes here are forever.

the append

Every mistake has to happen left of the line. Design so the line itself is boring.



Errata: entry 39

What happened

While expanding the production pipeline, a test package was written to the production log.

It is entry 39. It is still there. It always will be.

The package was never released publicly - but the entry is public, and permanent.

Published as errata, not an internal postmortem.

Other roads to a wrong entry

A change that only exists in production, so no test ever saw it

A pre-release build treated as if it were public

Ambiguous identity across variants: a genuine hash, under a label that does not say what it really is

The log cannot be rewritten to hide our own mistakes. That is the feature, working-as-intended.

https://developers.google.com/android/binary_transparency/google1p/errata

Safeguards that earned their place



Test-package containment

Only whitelisted packages can be written. An internal or test package that reaches the pipeline is refused, not logged. Entry 39 is why this matters.



Strict server targeting

The whitelist also decides which log a package goes to. No match, no write: an unknown package fails closed instead of landing in the wrong log.



Public releases only

Only production and canary tracks are logged. QA, alpha and beta builds are filtered out before they get anywhere near the log.



All or retry

If any part of a log update operation fails, the whole operation is redelivered and retried. A missing entry is a bug we can fix; a wrong entry is a fact we have to live with.

The audience has the score

An outside reader checked our Pixel log with only public data and our own open-source tools - and found two problems.



What they found

January 2026 Pixel factory images were missing from the log: 29 images across 15 devices.

Android 17 entries carried the generic system image fingerprint, not the device fingerprint, so on-device verification could not match them.

No inside access: our log, our factory images, a standard AVB tool.



What we did: append only

The missing images were appended. January now sits after later months, so the log itself records that it arrived late.

The generic entries stay, forever. Entries with the device fingerprint are being appended beside them.

The fingerprint source is fixed, and we are adding monitoring for missing releases.

Public log, public tools, public issue, public fix. That is transparency doing its job.

github.com/android/android-binary-transparency/issues/20

Witnessing: one performance, one audience

Append-only cannot rule out a split view. Witnesses can. You know this one.



New for us: every checkpoint now carries independent cosignatures.

Standard C2SP tlog-witness protocol - nothing bespoke. Who, and what happens when they do not answer: next slide.



Who's listening

Three independent operators

Public Witness Network: Google TrustFabric, Geomys, Glasklar Teknik

Ed25519 + ML-DSA-44

Post-quantum cosignatures, where the witness offers them

Policy is a file

C2SP tlog-policy format. Quorum: all three operators

We fail open

Witnessing never blocks publishing. Verifiers decide what to demand.

Closing the loop: Uraniborg

A log nobody checks is just an expensive database. Verification is the product.

1

Scan the device - enumerate installed APKs and hash them locally



2

Query the published log and verify the inclusion proof cryptographically



3

Check the checkpoint's cosignatures - is this the same log everyone else sees?



4

[Open source: github.com/android/android-binary-transparency/](https://github.com/android/android-binary-transparency/) --> uraniborg



Next: see it run - scan, look up, verify

Demo: Uraniborg

One emulated device, the public log, no inside access.



<https://github.com/billy-lau/android-binary-transparency/tree/experimental/uraniborg/UI/uraniborg>

What to take home

01

Pay for correctness upstream of the commit. Append-only systems have no eraser, so dedup, normalization and validation belong before the write.

02

Make identity explicit: one logical release, many binary hashes. Variant-heavy pipelines break every schema that assumes one artifact per version.

03

If you cannot test in staging, contain in production - and publish your mistakes. An operator who can quietly edit the log has not built a transparency log.

04

Do not be the only signer, or the only verifier. Cosigned checkpoints and an open-source auditor are what turn your promises into someone else's evidence.



Sostenuto

Anyone can strike the note. The engineering is in holding it.

GPAT | Mainline Modules | Errata | Uraniborg | Public Witness Network

Questions