

Bitwise Reproducible Kernel Builds - PoC

Joshua Lilly

jgl@meta.com

The claim

- A build is bitwise reproducible when the same source, config, and toolchain yield byte-for-byte identical output.
- Identical vmlinux
- Identical arch/x86/boot/bzImage
- Verifiable by sha256sum

For a normal kernel this is hygiene. For a Trusted Execution Environment it is foundational.

Why a TEE changes the stakes

A TEE measures the code it boots and reports a cryptographic hash.

That hash is only meaningful if someone can independently derive the expected value.

An auditor, WhatsApp, or an external reviewer should be able to take the published source and config, rebuild, and arrive at exactly the hash the hardware attests to.

The gap

Without reproducibility there is an unverifiable gap between "the source we published" and "the binary that is running."

Attestation degrades from a proof into a promise.

Reproducibility closes that gap and lets the trust chain be checked end to end, by anyone.

How we got here

- A year ago we were asked to make the TEE kernel build bitwise reproducible.
- The ask was a question, not a specification: is this achievable at all?
- Upstream documented individual sources of non-determinism.
- What it did not offer was an end-to-end result we could point at and copy.

Working without a guarantee

- An unknown number of divergence sources, each one invisible until the one before it is fixed.
- One bit of feedback per build: the hash matches, or it does not.
- No way to estimate how far away done was, or whether done existed.
- That is why the method mattered more than any individual fix. It made progress measurable while the endpoint was still unknown.

A stock kernel build is not deterministic

- Every build embeds things that vary run to run:
- Build timestamps
- Building user and host
- Git-derived version strings
- An archive of kernel headers (CONFIG_IKHEADERS)
- Ephemeral module-signing keys

Any one of these perturbs the final hash even when no source changed.

Upstream already has a checklist

Documentation/kbuild/reproducible-builds.rst

Source of non-determinism	Upstream remedy
Timestamps	KBUILD_BUILD_TIMESTAMP, SOURCE_DATE_EPOCH
Build user / host	KBUILD_BUILD_USER, KBUILD_BUILD_HOST
Absolute build paths	prefix-map via KCFLAGS / KAFLAGS
Module signatures	detached signatures published as sources, two-phase build
Structure randomisation	pre-generate scripts/basic/randstruct.seed
vDSO debug-info collisions	CONFIG_BUILD_SALT
Git state	setlocalversion, include/config/kernel.release

So was this enough?

The checklist tells you what the categories are.

It does not tell you which of the objects in your tree is biting you right now.

Configuration knowledge is not localization.

A failing build gives you one bit of feedback: the hash differs.

The insight

- The image hash is a deterministic function of the object files linked into it.
- If every .o in every subsystem compiles identically, the whole-image hash should be the same.
- A global failure therefore decomposes into a set of local ones.
- "The image differs" is intractable
- "These 3 of 4,800 objects differ" is a work item

The method: manifest diffing

1. Build, then hash every .o in the tree into a manifest
2. Build again against a copy of the previous build (t1/, build1/)
3. Diff the two manifests
4. Each surviving entry is one concrete lead
5. Fix, rebuild, re-diff; the list shrinks
6. Reproducibility is reached when the diff is empty
7. The whole harness: a dozen lines of shell plus a set comparison.

Three properties that make it work

- **Stopping condition.** Empty diff. Not a judgment call, not "looks the same."
Mechanically checkable in CI.
- **Monotone progress.** Every iteration strictly shrinks the set, so you always know whether you moved forward.
- **The diff is a pointer, not an answer.** kheaders.o differing does not say why. It says where to look. Localize first, diagnose second.

Where this sits next to existing tooling

When the artifact is a 4,800-object kernel image, that is the question you have to answer first.

Tool	Question it answers
diffoscope	How do these two artifacts differ?
reprotest	Which environmental variable causes it?
Manifest diffing	Which object differs?

Finding 1: kheaders.o

CONFIG_IKHEADERS embeds a timestamped archive of the kernel headers.

The archive is rebuilt every time, so the object never matches.

Disabled for the PoC.

Finding 2: version strings

CONFIG_LOCALVERSION_AUTO appends a git-derived suffix.

Turned off, with LOCALVERSION and BUILD_SALT left empty.

Generated files under init/ and arch/ pinned.

Finding 3: module signing

CONFIG_MODULE_SIG signs with a per-build ephemeral key.

Disabled, captured as a standalone remove_module_signing change.

Finding 4: GDB scripts

CONFIG_GDB_SCRIPTS generates non-deterministic content.

Disabled.

Finding 5: build environment

The build records the time, user, and host it ran on.

Pinned KBUILD_BUILD_TIMESTAMP, KBUILD_BUILD_USER=builder, KBUILD_BUILD_HOST=buildhost, and SOURCE_DATE_EPOCH.

On disabling module signing

The objection: you turned off a security feature to gain a security property.

Upstream's answer is a two-phase build: sign after the first build, publish the detached signatures, attach them in the second.

The key stops being a per-build secret and becomes a published input to verification.

Same shape as RANDSTRUCT: pin the seed and publish it.

Results

- Roughly 4,800 tracked objects. Empty diff. Identical sha256sum and md5sum for both vmlinux and bzImage.
- Deliberate boundaries:
- Built against vanilla origin/linus-upstream, not the internal TEE kernel, to isolate reproducibility from the out-of-tree patch stack
- A standard Meta devserver, not a pinned hermetic build image
- Features disabled that production will want back

To production

1. Hermetic build image with a pinned toolchain, Nix or a container
2. Extend manifest diffing to the internal patch stack
3. CI gate on a non-empty diff
4. Publish the verifier's inputs: config, randstruct seed, detached signatures

The method transfers. The config work does not.

Attestation should be a proof, not a promise.

Bitwise Reproducible Kernel Builds - PoC

Joshua Lilly

jgl@meta.com