NixOS Android Builder

Contents

Design Principles

With the goal of enabling offline, SLSA-compliant builds for custom distributions of Android AOSP, we set out to create a minimal Linux system with the following properties:

- **Portable** – Runs on arbitrary `x86_64` hardware with UEFI boot, that provides sufficient disk (>=250 GB) and memory (>=64 GB) to build Android.
- **Offline** – Requires no network connectivity other than to internal source-code and artifact repositories.
- **Ephemeral** – Each boot of the builder should result in a pristine environment; no trace of build inputs or artifacts should remain after a build.
- **Declarative** – All aspects of the build system are described in Nix expressions, ensuring identical behavior regardless of the build environment or the time of build.
- **Trusted** – All deployed artifacts, such as disk images, are cryptographically signed for tamper prevention and provenance.

We created a modular proof-of-concept based on NixOS that fulfills most of these properties, with the remaining limitations and future plans detailed below. Usage instructions can be found in ./user-guide.pdf.

Limitations and Further Work

- **aarch64 support** could be added if needed. Only `x86_64` with `UEFI` is implemented at the moment.
- **unattended mode** is not yet fully-tested. The current implementation includes an interactive shell and debug tools.
- **artifact uploads**: build artifacts are currently not automatically uploaded anywhere, but stay on the build machine until it is rebooted.. Integration of a Trusted Platform Module (TPM) could be useful here, to ease authentication to private repositories as well as destinations for artifact upload.
- **measured boot**: while we use Secure Boot with a platform custom key, we do not measure involved components via a TPM yet. Doing so would improve existing Secure Boot measures as well as help with implementing attestation capabilities later on.
- **credential handling** we do not currently implement any measures to handle secrets other than what NixOS ships out of the box.
- **higher-level configuration**: Adapting the build environment to the needs of custom AOSP distributions might need extra work. Depending on the nature of those customizations, a good understanding of `nix` might be needed. We will ease those as far as possible, as we learn more about users customization needs.

Used Technologies

- `NixOS` - the Linux distribution chosen for its declarative module system and flexible boot process.
- `nixpkgs` - the software repository that enables reproducible builds of up-to-date open-source packages.
- `qemu` - used to run virtual machines during interactive, as well as automated testing. Both help to decrease testing & verification cycles during development & customization.
- `systemd` - orchestrates both upstream and custom components while managing credentials and persistent state.
- `systemd-repart` - prepares signable read-only disk images for the builder and resizes and re-encrypts the state partition at each boot.
- **Linux Unified Key Setup (`LUKS`)** - encrypts the state partition with an ephemerally generated key on each boot.
- Various **build requirements** for Android, such as Python 3 and OpenJDK. The complete list is in the `packages` section of `android-build-env.nix`.

A complete **Software Bill of Materials (SBOM)** for the builder's NixOS closure can be generated from the repository root by running, e.g.:

```
nix run github:tiiuae/sbomnix#sbomnix -- .#nixosConfigurations.vm.toplevel
```

Major Components

The **NixOS Android Builder** is a collection of Nix expressions (a "nix flake") and helper scripts that produce a reproducible[1], ready-to-flash Linux system capable of compiling Android Open Source Project (AOSP) code. The flake pins `nixpkgs` to a specific commit, ensuring that the same versions of compilers, libraries, and build tools are used on every build. Inside the flake, a NixOS module describes the system layout, the `android-build-env` package, and the custom `fhsenv` derivation that provides conventional Linux file system hierarchy. This approach guarantees that the same inputs always generate the same output, making the build process deterministic and auditable.

Users with `nix` installed can clone this repository, download all dependencies and build a signed disk image, ready to flash & boot on the build machine, in a few simple steps outlined in README.md.

The resulting disk image boots on generic `x86_64` hardware with `UEFI` as well as Secure Boot, and provides an isolated build environment. It contains scripts for secure boot enrollment, a verified filesystem, and an ephemeral, encrypted state partition that holds build artifacts that cannot fit into memory.

Disk Image

A ready-made disk image to run NixOS Android Builder on a target host can be build from any existing `x86_64-linux` system with `nix` installed. Under the hood, the image itself is built by `systemd-repart`, using NixOS module definitions from `nixpkgs` as well as custom enhancements shipped in this repository.

Build Process

`systemd-repart` is called twice during build-time:

1. While building `system.build.intermediateImage`: A first image is built, it contains the `store` partition, populated with our NixOS closure as well as minimal `var-lib` partition. `boot` and `store-verity` remain empty during this step.

2. While building `system.build.finalImage`: Take the populated `store` partition from the first step, derive `dm-verity` hashes from them and write them into `store-verity`. The resulting `usrhash` is added to a newly built `UKI`, which is then copied to `boot`, to a path were the firmware finds it (`/EFI/BOOT/BOOTX86.EFI`).

3. The image then needs to be signed with a script outside a `nix` build process (to avoid leaking keys into the world-readable `/nix/store`. No `systemd-repart` is involved in this step. Instead we use `mtools` to read the `UKI` from the image, sign it and - together with Secure Boot update bundles, write it back to `boot` inside the image.

4. Finally, `systemd-repart` is called once more during run-time, in early boot at the start of `initrd`: The minimal `var-lib` partition, created in the first step above, is resized and encrypted with a new random key on each boot. That key is generated just before `systemd-repart` in our custom `generate-disk-key.service`.

[1] *Reproducible* in functionality. The final disk images are not yet expected to be *fully* bit-by-bit reproducible. That could be done, but would require a long-tail of removing additional sources of indeterminism, such as as date & time of build. See reproducible.nixos.org

Disk Layout

Partition	Label	Format	Mountpoint
00-esp	`boot`	`vfat`	`/boot`
10-store-verity	`store-verity`	`dm-verity hash`	`n/a`
20-store	`store`	`erofs`	`/usr`
30-var-lib	`var-lib`	`ext4`	`/var/lib`

- **boot** – Holds the signed Unified Kernel Image (`UKI`) as an `EFI` application, as well as Secure Boot update bundles for enrollment. The partition itself is unsigned and mounted read-only during boot.
- **store-verity** – Stores the `dm-verity` hash for the `/usr` partition. The hash is passed as `usrhash` in the kernel command line, which is signed as part of the `UKI`.
- **store** – Contains the read-only Nix store, bind-mounted into `/nix/store` in the running system. The integrity of `/usr` is verified at runtime using `dm-verity`.
- **var-lib** – A minimal, ephemeral state partition. See next section below.

Notably, the root filesystem (`/`) is, along with an optional writable overlay of the Nix store, kept entirely in RAM (`tmpfs`) and therefore not present in the image. There's also no boot loader, because the `UKI` acts as an `EFI` application and is directly loaded by the hosts firmware.

Ephemeral State Partition

The `/var/lib` partition is deliberately designed to be temporary and encrypted. Each time the system boots, a fresh key is generated and the partition is resized to match the current disk size. This ensures that sensitive build artifacts never persist beyond a single session, reducing the risk of leaking proprietary information or to introduce impurities between different builds.

Secure Boot Support

Secure Boot is enabled by generating a set of keys that are stored unencrypted in a local `keys/` directory within the repository. Users must protect these keys and back them up. When a new image is signed, Secure Boot update bundles (`*.auth` files) are created for each target machine. These bundles are stored unsigned and unencrypted on the `/boot` partition. On boot, we check whether whe are in Secure Boot setup mode and, if so, enroll our keys. If Secure Boot is disabled, we display an error and fail early during boot.

Custom FHS Environment

The builder image includes a custom builder for File Hierarchy Standard (`FHS`) environments.

It consists of a derivation that runs a python script, `fhsenv.py` to bundle together all libraries and binaries of declared packages (`nixosAndroidBuilder.fhsEnv.packages`), arranging them in one big `FHS` layout with `/bin` & `/lib` directories in the derivations output.

A mechanism to pin specific instances of packages which might be included multiple times inside the transitive dependency tree. See `nixosAndroidBuilder.fhsEnv.pins`.

The `fhsenv.nix` NixOS Module bind-mounts `/lib` and `bin` from the derivations output during runtime, while also setting default pins / packages, `$PATH` and adding a custom build of `glibc` for its dynamic linker, and a `FHS`-compatible build of `bash`.

That dynamic linker is configured to `/lib` instead of the standard Nix store paths. This setup mimics a conventional Linux environment, allowing the Android build system to function without modification.

Alternative approaches, such as `pkgs.buildFHSEnv`, `nix-ld` or `envfs`, were evaluated but found insufficient because they rely on individual symlinks that break when sandboxed bind-mounts are applied to `/bin` and `/lib` only, without having `/nix/store` in the sandbox.

Android Build Environment

The `android-build-env.nix` NixOS module uses the `fhsenv.nix` module described in the section above, to add all tools required by for an AOSP build. By using this module, developers can compile Android in a clean, reproducible environment that mimics a standard Linux installation.

It also adds 3 scripts, added for convinience:

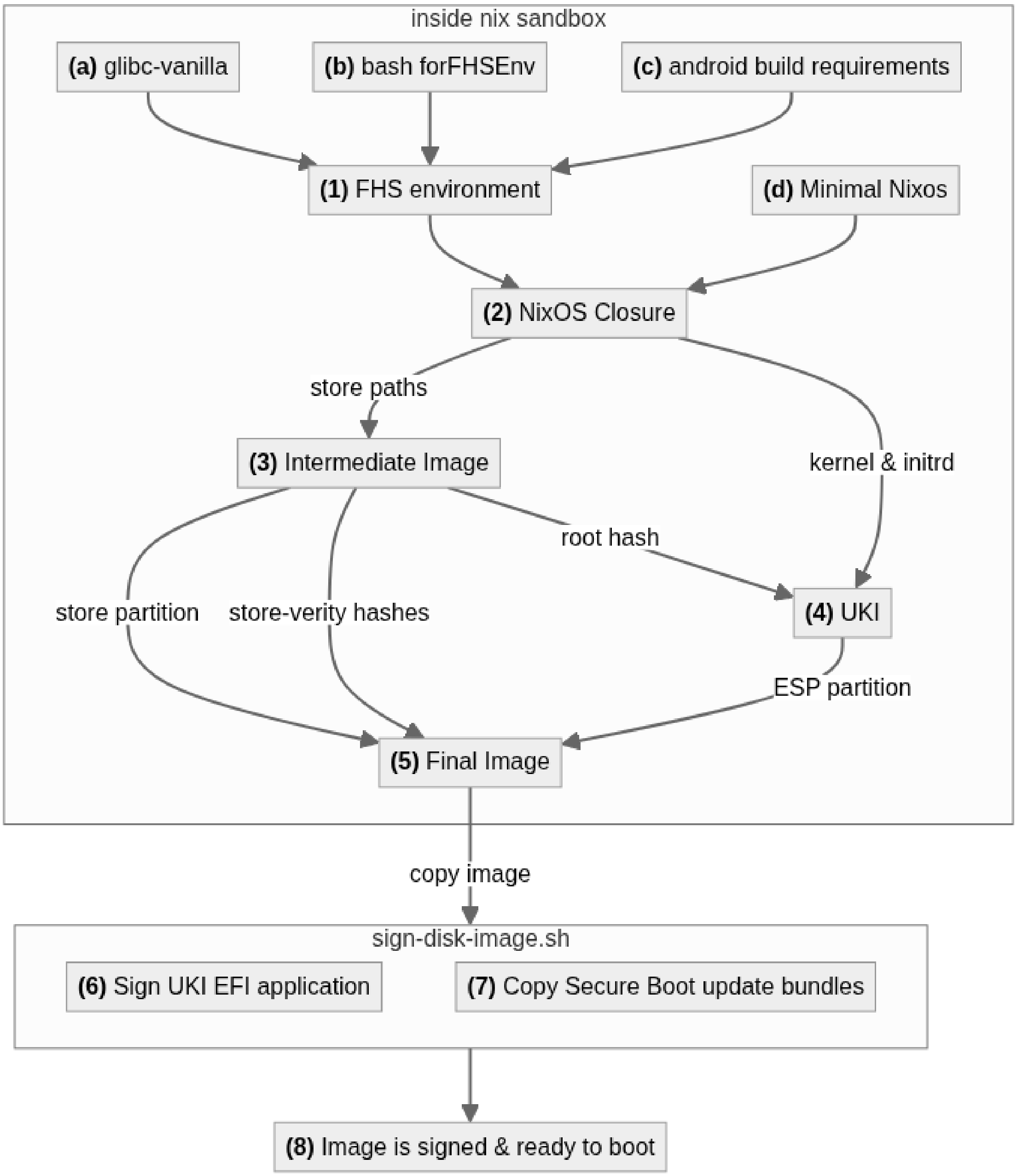
- `fetch-android` checks out the configured `repo` repository & branch, upstream AOSP's `android-latest-release` by default.
- `build-android` loads the shell setup, sets the configured `lunch` target and builds a given `m` target.
- `sbom-android` is a thin wrapper around `build-android` to run upstream's Software Bill Of Materials facilities.

Please refer to the options reference in user-guide.pdf.

Sequence Chart

Build-time

The following chart depicts a high-level overview on how the different components are assembled into the final disk image at build-time. A detailed description of the steps follows after the chart.



Description

- (1) We start by building an FHS environment in a derivation, as outlined above. Main components are:
  - (a) `glibc-vanilla` - NixOS glibc, but with a dynamic linker configured to search `FHS` paths, such as `/lib`, `/bin`, …
  - (b) `bash` with `forFHSEnv` set to `true`. NixOS bash does not include `bin` in `PATH` in empty environments. Built with `forFHSEnv` it does.

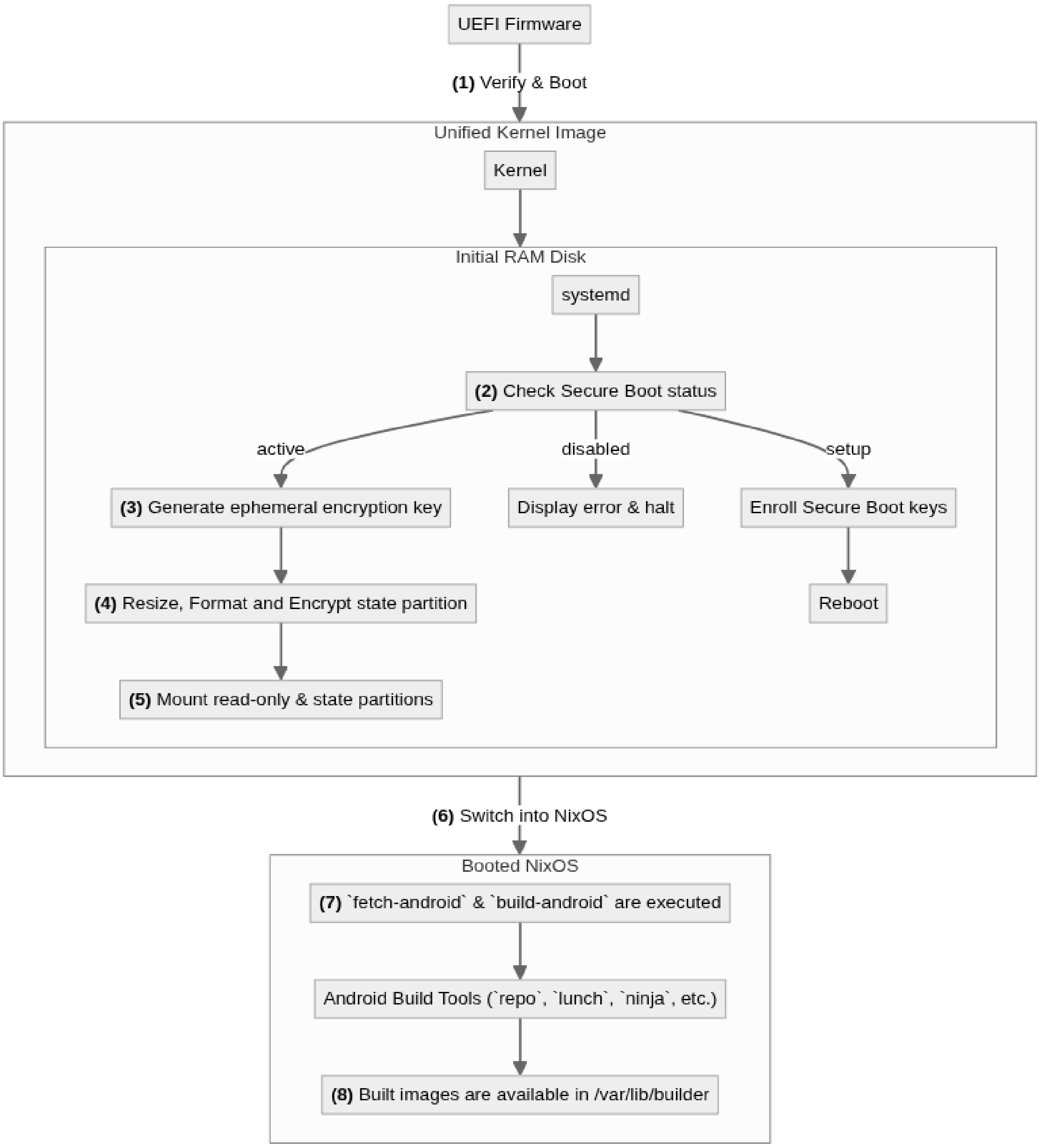
  - **(c)** Android build dependencies that are not shipped in-tree. `repo`, etc.
- **(2)** The NixOS closure (`system.build.toplevel`) is build, including **(d)** boot & system services as well as, the `fhsenv` derivation from the previous step.
- **(3)** First run of `systemd-repart` (`system.build.intermediateImage`):
  - Starts from a blank disk image.
  - Store paths from the NixOS closure are copied into the newly `store` partition.
  - `esp`, `store-verity` and `var-lib` are created but stay empty for the moment.
- **(4)** With a filled store partition, `dm-verity` hashes can be calculated. So we build a new `UKI`, taking kernel & initrd from the NixOS closure and adding the root hash of the `dm-verity` merkle tree to the kernels command line as `usrhash`.
- **(5)** Second run of `systemd-repart` (`system.build.finalImage`):
  - Starts from the intermediate image from step **(3)**.
  - The `store` and `var-lib` partitions are copied as-is.
  - `dm-verity` hashes are written to the `store-verity` partition.
  - The unsigned `UKI` from step **(4)** is copied into the `esp` partition.
  - With that being done, the image is built and contains our entire NixOS closure, including the `fhsenv`, in a `dm-verity`-checked store partition, as well as the `UKI` including `usrhash`.

All that's left to do, is to sign it and prepare it for Secure Boot. The `UKI` is not yet signed, as doing so inside the nix sandbox, might expose the signing keys. So the user is asked to copy the built image from the nix store to a writable location and execute `sign-disk-image.sh` on it. Usage is documented in user-guide.pdf. `sign-disk-image.sh` manipulates the `vfat` partition inside the disk image directly, in order to:

- **(6)** The `UKI` is copied to a temporary file, signed, and copied back into the `esp` again.
- **(7)** Secure Boot update bundles (`*.auth` files) are copied to the `esp` to ensure that `ensure-secure-boot-enrollment.service` can find them during boot.
- **(8)** We finally have a signed image, reado to flash & boot on a target machine.

Run-time

The following chart depicts a high-level overview on steps that run after the disk image has been booted on target hardware. A detailed description of the steps follows after the chart.



Description

1. The hosts EFI firmware boots into the Unified Kernel Image (`UKI`), verifying its cryptographic signature if secure boot is active. A service to check that Secure Boot is active runs early in the `UKI`s initial RAM disk (`initrd`).

2. `ensure-secure-boot-enrollment.service`, asks EFI firmware about the current Secure Boot status.

- If it is **active** and our image is booting succesfully, we trust the firmware here and continue to boot normally.
- If it is in **setup** mode, we enroll certificates stored on our ESP. Setting the platform key disables setup mode automatically and reboot the machine right after.

- If it is **disabled** or in any unknown mode, we halt the machine but don't power it off to keep the error message readable.

3. Before encrypting the disks, we run `generate-disk-key.service`. A simple script that reads 64 bytes from `/dev/urandom` without ever storing it on disk. All state is encrypted with that key, so that if the host shuts down for whatever reason - including sudden power loss - the encrypted data ends up unusable.
4. `systemd-repart` searches for the small, empty state partition on its boot media and resizes it before using `LUKS` to encrypt it with the ephemeral key from **(2)**.
5. We proceed to mount required file systems:
   - A read-only `/usr` partition, containing our `/nix/store` and all software in the image, checked by `dm-verity`.
   - Bind-mounts for `/bin` and `/lib` to simulate a conventional, FHS-based Linux for the build.
   - An ephemeral `/` file system (`tmpfs`)
   - `/var/lib` from the encrypted partition created in **(3)**.
6. With all mounts in place, we are ready to finish the boot process by switching into Stage 2 of NixOS.
7. With the system fully booted, we can start the build in various ways. The current implementation still includes an inteactive shell and 2 demo scripts which can be used as a starting point:
   - `fetch-android` uses Androids `repo` utility to clone the latest `AOSP` release from `android.googlesource.com` to `/var/lib/build/source`.
   - `build-android` sources required environment variables before building a minimal `x86_64` `AOSP` image.
8. Finally, build outputs can be found in-tree, depending on the targets built. E.g. `/var/lib/build/source/out/target/product/vsoc_x86_64_only`. Those are currently not persisted on the builder, so manual copying is required if build outputs should be kept.