technische universität
dortmund

Bachelorarbeit im Fach Informatik

A Full-Source Bootstrap for NixOS

Wire Jansen
27. Oktober 2025

Gutachter:
Prof. Dr.-Ing. Peter Ulbrich
Alwin Berger, M.Sc.

Technische Universität Dortmund
Fakultät für Informatik
Arbeitsgruppe Systemsoftware (LS-12)
https://sys.cs.tu-dortmund.de

ABSTRACT

Even when compiled from source, executables are at most as trustworthy as the compiler that produced them. Considering that the compilers commonly used to build Linux-based operating systems are usually compiled with existing binary versions of themselves, their trustworthiness can not be guaranteed. The process of building a compiler with a different compiler, thus breaking the loop, is called "bootstrapping".

In this thesis, we present our implementation of a full-source bootstrap for the NixOS Linux distribution. We build a Linux environment with the Nix package manager from a minimal, hand-auditable binary seed, and then use it to install NixOS. For this, we use a version of Nixpkgs—the package set on which NixOS is based—that we also modified to be bootstrapped from a hand-auditable binary seed. Although there are compilers in Nixpkgs that we can not build from the bootstrap seed yet, the result is a NixOS installation, most parts of which are fully compiled from source. Finally, we discuss the practicality of the full-source bootstrap and how it can be improved upon so that every NixOS user can benefit from the advances in trustworthiness we have achieved.

KURZFASSUNG

Selbst wenn sie aus Quelltext kompiliert wurden, können ausführbare Programmdateien maximal so vertrauenswürdig sein wie der Compiler, durch den sie erzeugt wurden. Die Compiler, die zum Übersetzen Linux-basierter Betriebssysteme üblicherweise verwendet werden, werden für gewöhnlich mittels einer existierenden Kopie ihrer selbst kompiliert. Aufgrund der Tatsache, dass diese als ausführbare Dateien vorliegen, kann ihre Vertrauenswürdigkeit nicht garantiert werden. Der Vorgang, einen solchen selbstübersetzenden Compiler mit Hilfe eines anderen Compilers zu bauen, um diesen Kreis zu durchbrechen, wird als „Bootstrapping" bezeichnet.

In dieser Arbeit präsentieren wir unsere Umsetzung eines vollständigen Bootstraps für die Linuxdistribution NixOS. Wir bauen eine Linux-Umgebung mit dem Paketverwaltungsprogramm Nix aus einer minimalen, menschenüberprüfbaren Ausgangsbinärdatei und benutzen diese, um NixOS zu installieren. Zu diesem Zweck verwenden wir eine modifizierte Version von Nixpkgs—dem Paketsatz, auf dem NixOS basiert—welche ebenfalls aus einer menschenüberprüfbaren Ausgangsbinärdatei erzeugt wird. Ungeachtet der Tatsache, dass Nixpkgs Compiler enthält, welche wir noch nicht aus der Ausgangsbinärdatei erzeugen können, ist das Resultat eine NixOS-Installation, die, zum überwiegenden Teil, vollständig aus Quelltext erzeugt wurde. Schließlich erörtern wir die Zweckmäßigkeit des vollständigen Bootstraps und wie dieser verbessert werden kann, sodass alle Anwender*innen von NixOS von den Fortschritten in Sachen Vertrauenswürdigkeit, die wir erreicht haben, profitieren können.

CONTENTS

1 INTRODUCTION

To use any computer system effectively, the user must trust that the software running on it will act in their best interests. With proprietary software, which is available only in binary form, this trust has to be awarded uncritically, based solely on the vendor's claims and reputation.

Free and open-source software (FOSS) is generally considered more trustworthy because the source code used to compile the executables is publicly available, allowing users and independent security researchers to review it. Many FOSS projects are developed fully in the open, allowing anyone to track and investigate every change to their source code. In theory, the user could even read and then compile it on their own hardware, fully absolving them from having to trust the author's or any third party's claims about the software.

1.1 Binary Packages

On Linux systems, software is often installed through a package manager. Usually, this is the one that ships with the user's distribution of choice. A lot of the time, the software included in a distribution's official package repository[1], as well as the build scripts that were used to create the packages, are FOSS. The user, on the other hand, only downloads binaries built by the maintainers of the package repositories. However, popular package managers like *the Debian Package Manager (dpkg)* and *the RPM Package Manager (RPM)* offer no easy way to verify that the packages were actually built from the available source code, requiring the user to trust the packages' supplier, just like with proprietary software.

Source-based package managers, like Gentoo's *Portage* [1], solve this by downloading the package's source code and compiling it locally on the user's machine instead. While this solves the problem of having to trust someone else with building the executables, local compilation also significantly increases the time and computing power required to install a new package, thereby making source-based package managers less convenient to use than binary-based ones.

Nix, a source-based package manager that we will introduce in greater detail in Section 2.1, circumvents this inconvenience by using a binary cache. By default, instead of building everything locally, Nix queries the configured binary caches to check if they contain a prebuilt copy of the package and, if so, uses that instead. In this configuration, Nix behaves almost like a binary-based package manager. Unlike with those, the retrieval of prebuilt packages ("substitution" as Nix calls it) can be deactivated if the user does not want to trust the binary cache operators. Even if it is turned on,

[1] the place from where the package manager obtains the software

Nix can easily rebuild any package locally and compare the result to the one obtained from a binary cache[2].

1.2 Trusting Compilers

To be able to ultimately trust that the produced binaries behave exactly as described by the source code they were built from, building just the package itself locally is not enough: As K. Thompson demonstrated in his 1984 Turing Award lecture "Reflections on Trusting Trust" [2], any executable, no matter how trustworthy its source code may be, can only ever be as trustworthy as the compiler that was used to build it. That is because a malicious compiler could be created that injects malicious code into the executables it produces.

Thompson suggests that this could result in a virtually undetectable trojan when applied to a self-hosting compiler. A compiler is self-hosting if it can compile itself. Self-hosting compilers are usually built using an earlier version of themselves. Such a compiler could be modified to detect when it is compiling itself, and inject the trojan code into the new executable. Once a first infected compiler has been built, all subsequent rebuilds will also carry the trojan, even if it is removed from the source code afterward. While Thompson used the original UNIX C compiler for his demonstration in the lecture, a significant number of the compilers used to build the software running on modern computer systems, like *gcc*, *rustc*, and *go*, are self-hosting and therefore vulnerable in the same way.

Circumventing this kind of attack requires building the compiler using another compiler. The process of building a self-hosting compiler "from scratch" without using a previous version of the compiler is called "bootstrapping". Because the bootstrap compiler could be similarly modified to infect the other compiler when used for the bootstrap, it, too, needs to be bootstrapped from another compiler, creating a bootstrap chain. Ultimately, this means that a trustworthy system has to be bootstrapped from a compiler that is written in machine code by hand and therefore can be executed directly, without requiring a compiler to build it. That is what is called a "full-source bootstrap"[3], as everything is built from human-readable sources.

Because of the way Nix is designed, packages can only use programs from other Nix packages in their build scripts. That means it is not possible to use a compiler already present on the host within a Nix build. Cyclic dependencies are not possible either. Therefore, the entire dependency tree of *Nixpkgs*, the Nix package repository, is rooted in a single package, `bootstrapTools`, which contains an initial set of prebuilt binaries that are used to bootstrap the rest of the package set [4, `/pkgs/stdenv/linux`], [5, p. 177], [6, pp. 21–24]. This makes NixOS, the Linux distribution built from the packages in Nixpkgs, an ideal candidate for attempting the full-source bootstrap of a complete Linux distribution.

[2]`nix-build --check`

[3]The term "full-source bootstrap" was coined by the Guix project [3]. See Section 6.1

1.3 Research Questions

This thesis answers the following questions:

- **RQ1** Is it possible to construct a full-source bootstrap chain for the Nix package manager?
- **RQ2** Is it possible to construct a full-source bootstrap chain for Nixpkgs by replacing the `bootstrapTools` package?
- **RQ3** How can these two bootstrap chains be combined to create a full-source bootstrap for NixOS?
- **RQ4** Can the bootstrap chain be executed fully offline?
- **RQ5** Is it possible to create a NixOS ISO image that can be used to install the bootstrapped NixOS on other computers without having to run the full bootstrap on them first?
- **RQ6** How can the bootstrap be adapted to architectures beyond x86-based systems?
- **RQ7** Does the bootstrap chain still include any binaries besides the initial bootstrap seeds?

BACKGROUND 2

2.1 Nix

As already touched on in Section 1.1, Nix is a purely functional, source-based package manager. It was created by E. Dolstra and is described in his 2006 doctoral thesis *The Purely Functional Software Deployment Model* [5].

2.1.1 The Nix Store

Unlike traditional package managers, Nix does not install any package files to the directories specified by the Filesystem Hierarchy Standard (FHS) [7]. Instead, all downloaded packages[4] are kept in a single central location on the file system called the *Nix store*, usually located at `/nix/store`.[5] [5, p. 19]

The packages in the Nix store are stored unpacked under paths of the form `<hash>-<name>`, where `<name>` is the package's name as specified in its definition and `<hash>` is a cryptographic hash calculated from the package's *inputs*.[6] The inputs are the package's build instructions, which include the store paths for all dependencies of the package. This way, every Nix package has a globally unique store path. If anything about the package definition changes, its store path changes, too, causing a rebuild. If the calculated store path already exists, the build is skipped, so that if a package definition is modified and later reverted, the old version does not have to be built twice. Because the hash is calculated from data that includes the store paths of all dependencies, a change to a package's store path causes the store paths of all packages that depend on it to change as well, ensuring that all dependents are rebuilt against the changed version. [5, pp. 19–21]

Due to the fact that all Nix packages are linked directly against their dependencies' store paths, their executables can be used directly from the Nix store. That is why Nix does not need to install packages into global directories like other package managers do. Because of this architecture, Nix can keep multiple versions of a package in its store at the same time, allowing it to install two programs side-by-side, even when they depend on two different versions of the same dependency that would otherwise be incompatible. [5, pp. 21, 23–24]

[4]The original doctoral thesis [5, p. 19] uses the term *component* for what we refer to as a package here. Current documentation [8] uses *package* instead, and we will follow that in this thesis.

[5]Even though Nix does not mandate the Nix store be in that location, packages built on one installation are only compatible with Nix installations using the same store location. Because most users want to use official binary cache, it is not practical to diverge from the default.

[6]As an example, a full store path for `bash` looks like this:
`/nix/store/aqbddpi6p0bjfdlgswjry90n3sgjsqsy-bash-interactive-5.2p37`

By default, Nix does not delete anything from the Nix store, unless the user manually triggers a garbage collector run. When this happens, all store paths that are not referenced as run-time dependencies by any package registered as a garbage collector root are deleted. [5, p. 124] Among others, all packages explicitly installed into the system or user environment are automatically referenced by a garbage collector root.

Even though packages are kept unpacked in the Nix store, Nix still includes a bespoke package file format: Nix Archives (NARs). [5, p. 92] It is primarily used for binary caches, which are directories containing NAR files, usually served via HTTP. When Nix is instructed to build a package, it first checks if its store path is already present locally. If not, it checks if the store path is available on one of the configured *substituters*. These can be either other Nix stores (e.g., on another machine accessed via SSH) or binary caches. [8, ch. 8.6.1] If the store path was found on a substituter, Nix downloads (and, in case of a binary cache, unpacks) it to the local Nix store. Only if it cannot be substituted is the package built from scratch.[7]

2.1.2 Nix, the Programming Language

Nix packages are defined using expressions written in Nix's domain-specific language (DSL), which is also called *Nix*. To avoid confusion between Nix, the package manager/interpreter, and Nix, the programming language, we will always refer to the latter as the *Nix DSL* or *Nix Language* in this thesis.

The Nix DSL is a lazily evaluated, purely functional programming language. In addition to types also found in other functional programming languages — numbers (integer, float), strings, booleans, lists, records (called *attribute sets (attrsets)*) — the Nix DSL also has the special types *path* and *string with context*.[8] [5, pp. 25, 62–63, 66–67, 71, 73], [8, ch. 5.1]

Path literals can be used to refer to file system locations in place of strings. Relative path literals are resolved relative to the location of the `.nix` file they appear in. When a path is combined with another string—for example, when used in a build script via string interpolation—the file or directory it points to is copied to the Nix store, producing a string with context. [5, p. 67], [8, ch. 5.1.1]

Strings with context behave exactly like regular strings in most circumstances. The context, accessible through the built-in function `getContext`, associates the string with one or more store paths. When two strings with context are combined, their contexts are merged. Nix uses string context to track the packages' build-time dependencies.[9] [8, ch. 5.1.1] Run-time dependencies are detected automatically as well; however, this happens at build-time, not during evaluation: After the build has been completed, Nix scans the file(s) in the resulting store path for occurrences of other store paths. The paths that are found this way become the package's run-time dependencies. [5, pp. 23–24]

The central element of the Nix Language is the function `derivation`. When it is evaluated, Nix creates (*instantiates*) a *store derivation* from the function arguments and returns a *derivation*. A store derivation is a `.drv` file in the Nix store that contains a serialized representation of everything needed to perform the actual build. The package can be built (*realized*) from the store derivation without evaluating any Nix DSL code.[10] [5, p. 39]

[7]Unless the package explicitly requests being built by setting `allowSubstitutes = false`. [8, ch. 5.4.1.1]

[8]According to [5, p. 67], there is also a "URI" type. While the syntax for it still works, it evaluates to a regular string on modern versions of Nix.

[9]String context was introduced in Nix 0.10 [8, ch. 14.53] and is therefore not mentioned in the doctoral thesis. Instead, paths pointing to other derivations had to be created using the—since removed—*subpath* operator. [5, pp. 72–73]

[10]`nix-store --realise /nix/store/<hash>-<name>.drv`

The derivation returned from the call to `derivation` is an attribute set containing strings with context that link it to the store derivation. An example of a minimum viable `derivation` call is shown in Listing 1.

```nix
let
  pkgs = import <nixpkgs> { system = "x86_64-linux"; };
in
derivation {
  name = "example";
  builder = "${pkgs.bash}/bin/bash";
  args = [ "-c" "echo hello world > $out" ];
  system = "x86_64-linux";
}
```

Listing 1: Minimum viable call to `derivation`

In this example, we create a package called "example" that contains a single file with the text "hello world". Packages are built by running the program specified in `builder` (the `bash` package from the 64-bit x86 version of Nixpkgs in this case) with the arguments specified in `args`. The `system` attribute tells Nix on which platform the package can be built. Aside from a set of special cases, all other attributes are passed to the builder as environment variables. [5, p. 28], [8, ch. 5.4.1, 5.4.1.1]

When evaluated, the Nix DSL code in Listing 1 produces a derivation that looks like this:

```
«derivation /nix/store/3abxnap3n654yfxdyccjwp37c6gjj9m4-example.drv»
```

Listing 2: Stringified derivation

As stated before, a derivation is actually just an attribute set. Nix detects that this attrset is a derivation because it contains `type = "derivation";` [5, p. 101] and uses this special syntax when printing it. Internally, the derivation looks like Listing 3.

To ensure that package builds behave as much as possible like functions in a purely functional programming language, whose result depends purely on their inputs, they are executed in a sandbox, isolated from the host system.[11]. Besides preventing the builder from accessing host files, it also blocks network access. It is possible to supply the hash of the build output to the `derivation` call to create a fixed-output derivation (FOD). In this case, the network restriction is lifted because the file is checked against the supplied hash and can therefore still be considered pure. The hash in the path of a FOD is calculated from the supplied hash rather than the derivation inputs, so that different FODs with the same hash produce the same store path, and it only has to be built once. [5, p. 106], [8, ch. 8.6.1]

Because the Nix DSL is lazily evaluated, only the derivations that are actually relevant for the current operation (e.g., building a specific package) are evaluated and therefore instantiated. [5, pp. 62–64] This is especially relevant in the context of large package sets like Nixpkgs. Without lazy evaluation, Nix would have to evaluate and instantiate all packages in Nixpkgs every time it is imported.

[11]The build sandbox has been introduced in Nix 0.11 [8, ch. 14.51] and is therefore not mentioned in the original thesis. However, the idea of trying to prevent host-system data from leaking into the build process has been discussed in [5, pp. 179–180].

```nix
{
  all = [ «derivation /nix/store/3ab...9m4-example.drv» ];
  args = [ "-c" "echo hello world > $out" ];
  builder = "/nix/store/aqb...qsy-bash-interactive-5.2p37/bin/bash";
  drvAttrs = {
    args = [ "-c" "echo hello world > $out" ];
    builder = "/nix/store/aqb...qsy-bash-interactive-5.2p37/bin/bash";
    name = "example";
    system = "x86_64-linux";
  };
  drvPath = "/nix/store/3ab...9m4-example.drv";
  name = "example";
  out = «derivation /nix/store/3ab...9m4-example.drv»;
  outPath = "/nix/store/n1x...s2k-example";
  outputName = "out";
  system = "x86_64-linux";
  type = "derivation";
}
```

Store paths have been shortened for readability.

Listing 3: Derivation attrset

2.1.3 Nixpkgs

Nixpkgs is the Nix project's official collection of Nix package definitions. It is provided as a single git repository. [4] With over 120000 packages,[12] it is the largest and most up-to-date package repository tracked by the indexing website *repology*. [9]

While the build definitions themselves are FOSS, the packaged software includes both FOSS and proprietary programs. By default, however, evaluating a proprietary package's definition produces an error. It must be explicitly allowed by the user, either globally or for individual packages. We do not set that option and can therefore assume that everything we use from Nixpkgs is FOSS and can be built from source. [10, `#sec-allow-unfree`]

2.1.3.1 `stdenv`

A central element of Nixpkgs is `stdenv`, the *standard environment*. It consists of a set of tools commonly needed for building software, and fulfills a similar role to packages like `build-essential` on Debian or `base-devel` on Arch Linux. [5, pp. 174–175]

[12]There are 129435 packages in the 25.05 version of Nixpkgs at the time of writing (2025-10-02). The current count can be obtained with this command:

```
curl -sL 'https://channels.nixos.org/nixos-25.05/packages.json.br' | brotli -d | jq -r '.packages | keys | .[]' | wc -l
```

Unlike those, `stdenv` is not installed onto the system. It is used through the function `stdenv.mkDerivation`, which is a wrapper around the built-in `derivation` function. It accepts all arguments accepted by `derivation` and can therefore be used as a drop-in replacement. `mkDerivation` extends upon the built-in `derivation` by adding the packages contained in `stdenv` to the build environment and providing a default builder, called the *generic builder*. [5, pp. 27, 175]

The *generic builder* is used when no other builder is specified. It is a shell script executed by `bash`. When using the generic builder, the build process is split into the *configure*, *build*, *check*, and *install* phases. Each phase has a default value intended to cover the common `./configure && make && make install` build process shared by many, especially C, programs. If the default script for a phase does not work for a given package, it can be selectively overwritten. Using the generic builder makes the package definitions similar to those of other package managers, like Arch Linux's `PKGBUILD` files or the `pass*.sh` files from *live-bootstrap*, which will be introduced in Section 2.3. Virtually all packages in Nixpkgs use `mkDerivation` and the generic builder; as a result, they depend on `stdenv`. [5, pp. 175–176]

example.nix — Nix

```nix
{ stdenv, fetchurl }:
stdenv.mkDerivation {
  name = "example";
  src = fetchurl {
    url = "<url>";
    sha256 = "<hash>";
  };

  configurePhase = ''
    ./configure --prefix=$out
  '';

  buildPhase = ''
    make
  '';

  installPhase = ''
    make install
  '';
}
```

A Nix DSL function that is passed `stdenv` and `fetchurl` as arguments and returns a derivation for the `example` package.

a) Nix

example/sources

```
<url> <hash> example.tar.gz
```

example/pass1.sh — Bash

```bash
src_configure () {
    ./configure
}

src_compile () {
    make
}

src_install () {
    make DESTDIR="${DESTDIR}" install
}
```

A `sources` and `pass1.sh` file that build the `example` package as a live-bootstrap step. The package name is infered from the directory name.

b) live-bootstrap

Both formats can build this example package with the defaults for their respective phases, meaning that these files could be considerably shorter. The commands are specified explicitly for illustration purposes.

Listing 4: Comparison of buildscript formats

To add packages to the build environment in addition to the ones from `stdenv`, `mkDerivation` accepts a set of options, the most commonly used of which are `buildInputs` and `nativeBuildInputs`. This split allows Nixpkgs packages to be cross-compiled easily. The `stdenv` derivation has three inputs that define the platform on which builds are executed and the platform they target. The names and purpose of the inputs are the same as the options GNU software, including the GNU Compiler Collection (GCC), uses for this purpose:

- `buildPlatform` is the platform on which the build is executed.
- `hostPlatform` is the platform on which the produced binaries can be executed. It diverges from `buildPlatform` during cross-compilation.
- `targetPlatform` is the platform the produced binaries target. It diverges from `hostPlatform` when building a cross-compiler.

All packages that need to be built for the host platform (e.g., libraries) must be placed in `buildInputs`, whereas all packages that need to be executed during the build process and for that reason need to be able to run on the build platform (e.g., compilers) must be placed in `nativeBuildInputs`. [10, `#ssec-cross-platform-parameters`, `#ssec-stdenv-dependencies-propagated`]

To ensure the purity of the build environment, the packages in `stdenv` are built using Nix, as well, thus creating a bootstrapping problem: Because Nix does not support cyclic dependencies, it is not possible to use the `stdenv` packages to build a new `stdenv` directly in the way it would be done with other package managers — by relying on the already globally installed versions of the tools. To overcome this, Nixpkgs uses the package `bootstrapTools`, which contains precompiled versions of the programs needed to create a minimal `stdenv`. These prebuilt binaries are downloaded as a FOD and then used to bootstrap the "real" `stdenv` used throughout Nixpkgs. [4, `/pkgs/stdenv/linux`]

The files that make up the `bootstrapTools` package are themselves built from Nixpkgs.[13] In that way, Nixpkgs is similar to a self-hosting compiler: A working version of Nixpkgs is required to build Nixpkgs.

2.1.3.2 Helper Functions

Aside from `mkDerivation` and its generic builder, Nixpkgs also provides other helper functions to create common kinds of derivations. Among these *trivial builders* are functions for creating a single file containing a specified string (`writeText`), creating a shell script in `/bin` (`writeShellScriptBin`), and creating a single-file C program (`writeCBin`). [4, `/pkgs/build-support/trivial-builders`]

The *fetchers* are helper functions used to create derivations that download files from the internet. They include `fetuchurl` for downloading a single file and `fetchgit` for downloading a specific commit from a git repository. Other fetchers also perform post-processing on the downloaded files. `fetchzip`, for example, extracts the archive after downloading, and `fetchpatch` normalizes the downloaded patch by removing everything that's not required for applying it (e.g., comments). All fetchers in Nixpkgs produce FODs, which means Nix verifies the integrity of the downloaded files. [4, `/pkgs/build-support/fetch*`]

The fetchers in Nixpkgs are not the only way to download files when using Nix. That would be a problem for bootstrapping Nixpkgs, because the fetchers it provides depend on programs like `curl` and `git` to perform the downloads. Nix has built-in versions of some fetchers, including `fetchurl`

[13] [4, `/pkgs/stdenv/linux/make-boostrap-tools.nix`]

and `fetchgit`. These support a subset of the Nixpkgs fetchers' API but differ in significant ways: Even though the result is a store path, too, the built-in fetchers do not create derivations. Unlike the Nixpkgs fetchers, where the download is performed by a program when building the derivation they produce, the built-in fetchers are programmed so that Nix itself performs the download the moment the call to the fetcher is evaluated. They also do not check whether the output path already exists and proceed with the download anyway. [10, `#chap-pkgs-fetchers`]

A special case is the fetcher implemented in `<nix/fetchurl.nix>`, a file included with the Nix source code. It implements a third API-compatible variant of `fetchurl`. This variant behaves like the Nixpkgs fetchers: It creates a FOD that downloads the file when it is built. What sets it apart is that it uses Nix's built-in downloading capabilities and therefore does not depend on any package. [11, `/src/libexpr/fetchurl.nix`]

In addition to packages and build helpers, Nixpkgs includes the standard library `lib`. It extends the built-in functions with a set of functions written in the Nix Language, serving a similar purpose as, e.g., Haskell's *prelude*. [4, `/lib`] Finally, Nixpkgs contains the source code for NixOS.

2.1.4 NixOS

NixOS is a Linux distribution that is built completely from Nix packages. [4, `/nixos`], [6]
The entirety of NixOS, including installed programs and activated services, along with their configuration, is managed through a central, modular configuration system. Modules, including the user-supplied system configuration[14], are written in the Nix DSL. Nixpkgs contains modules for many common services that can be used in any NixOS configuration without requiring explicit import. [4, `/nixos/modules`]

Evaluating a NixOS configuration returns an attrset containing the derivations used to build the configuration under the `config.system.build` attribute. The derivations describe different parts of the system, like the kernel and the contents of `/etc`. Most importantly, `config.system.build` contains the `toplevel` derivation, which provides everything that is needed to install the configuration. Among other things, `toplevel` contains the activation script (`activate`) that applies the configuration to the operating system (OS), and the `bin/switch-to-configuration` script that is used to install a new configuration and activate it either immediately or on the next reboot. [12, `#sec-building-parts`, `#sec-switching-systems`]

Because the Nix store can contain an arbitrary number of versions of the same package, the OS's previous state is preserved even when a new configuration is activated. All versions of the system configuration are registered as garbage collector roots to prevent Nix from deleting them or the packages installed by them, allowing NixOS to be rolled back to previous configurations, as long as the user does not explicitly prompt their disposal. Every time NixOS boots, it runs the activation script of the current configuration, allowing rollbacks to be performed directly in the bootloader. [12, `#sec-rollback`, `#sec-nix-gc`]

Due to the fact that on NixOS everything is contained in the Nix store, the usual FHS [7] directories are not populated. For that reason, binaries built for other Linux distributions cannot be executed on NixOS without additional measures. [6, p. 70]

[14]Usually located in `/etc/nixos/configuration.nix`.

2.2 Aux Foundation

The Aux project [13] aims to build an alternative to the Nixpkgs-centric Nix ecosystem. Their package set is bootstrapped from a hand-auditable 256-byte binary seed. This bootstrap chain is called *Aux Foundation*. [14], [15, `/POSIX/x86/hex0-seed`]

Currently, a working version is only available for 32-bit x86-based computers running Linux (or `i686-linux` as Nix calls this platform). *Aux Foundation* produces a set of packages that is similar to Nixpkgs' `stdenv`. Instead of *glibc*, it uses *musl* as its C standard library. Besides that, it does not include the complete set of packages in `stdenv` and therefore cannot be used as a drop-in replacement.

2.3 live-bootstrap

While a bootstrap chain like *Aux Foundation* may be able to build a package set without relying on any existing compiler, it still requires an OS (and in the case of *Aux Foundation*, a Nix installation) to run on. The OS itself might be compromised and could tamper with the files or the build process. As a consequence, to fully trust the bootstrapped binaries, the OS on which the build runs needs to be trustworthy as well.

The *live-bootstrap* [16] project solves this by bootstrapping an entire minimal Linux distribution. The seed for *live-bootstrap* is a hand-auditable 512-byte[15] binary, similar to the one used by *Aux Foundation*. Instead of a Linux executable, the binary seed used by *live-bootstrap* is bootable on its own, without requiring an OS. Like *Aux Foundation*, *live-bootstrap* currently only works on 32-bit x86-based computers and uses *musl* instead of *glibc*.

To run the bootstrap, the seed is placed on a disk, along with the source code for the programs to be bootstrapped. When a computer boots from this disk, it begins to execute the bootstrap chain. First, it bootstraps the Tiny C Compiler (tcc), which is used to build *fiwix* [18], a small UNIX / Linux-like kernel. After starting *fiwix*, the bootstrap chain continues by building GCC and using it to build Linux. Finally, on Linux, a small set of userland tools and the Grand Unified Bootloader (GRUB) are built and installed, resulting in a bootable Linux system. [16, `/parts.rst`]

live-bootstrap comes with its own simple package manager written in bash, which is used to define the build scripts for the bootstrapped programs. An example of such a build script is shown in Listing 4. Before the first bash has been built, *kaem*, a very simple shell script interpreter, is used instead. Each package can define multiple build scripts, named `pass1.sh`, `pass2.sh`, and so on, which are used when the package is built multiple times. That is a requirement in a bootstrapping scenario like this because some programs can only be built with additional functionality if another package is present, which might itself depend on the program. In such cases, the program is built once with the feature disabled, used to build the other package, and then rebuilt again with the feature enabled. [16, `/steps/helpers.sh`]

The bootstrap chain itself is defined in a custom manifest format, which is translated into scripts by a C program during the bootstrap's initial stages. Each line in the manifest contains an operation, a parameter, and, optionally, a condition that must be met for that line to be executed. The available operations include `build`, which builds the package passed as the parameter, `improve`, which runs a script from the `improve` directory, and `jump`, which runs a script from the `jump` directory and, before that, sets up `/init` so that the bootstrap may be resumed by executing it. The `jump` operation is used

[15]This is a full bootsector, including padding and a MBR partition table. The actual program is shorter than 512 bytes. [17, `/builder-hex0-x86-stage1.hex0`]

to execute a newly built kernel that, after loading, runs `/init` and resumes the bootstrap. [16, `/steps/manifest`, `/seed/script-generator.c`]

To ensure the reproducibility of the bootstrap, the `SHA256SUMS.pkgs` file contains hashes of the build results for all packages. After a package is built, the result is checked against the stored hash. [16, `/steps/SHA256SUMS.pkgs`]

The *live-bootstrap* repository contains a Python script that automates the bootstrap setup. The script has four modes: In the *chroot* and *bwrap* modes, it runs the bootstrap in a sandboxed environment, directly on the host OS. In *bare-metal* mode, the script produces disk images intended for use with a physical computer. In *QEMU* mode, it creates disk images, as in *bare-metal* mode, intended for use with a virtual machine (VM). After creating the images, it starts a QEMU VM in which the bootstrap is executed. The difference between the images created in the *bare-metal* and *QEMU* modes is that, while the *bare-metal* images display the log messages on the screen, the *QEMU* images use a serial port instead. [16, `/rootfs.py`]

Even though the bootstrapped OS is minimal and not suitable for general-purpose use, it is a valuable base for bootstrapping other software.

IMPLEMENTATION 3

Overall, our goal for this thesis is to show that it is possible to create a full-source bootstrap that starts with only source code and (ideally) a single human-auditable binary seed. For the purpose of simplifying development, we split the bootstrap chain into two distinct parts:

- The *outer bootstrap* builds a Linux OS with a Nix installation from a bootable binary seed.
- The *inner bootstrap*, on the other hand, assumes an existing, working Nix installation running under an existing Linux kernel and builds a NixOS *toplevel* derivation from a binary seed that can be executed inside the Nix build sandbox.

Both parts can be developed independently. The outer bootstrap does not require anything that the inner bootstrap provides. A copy of Nixpkgs is needed to validate that the bootstrapped Nix is functional, but an unmodified Nixpkgs is sufficient. Even though the inner bootstrap requires the result of the outer bootstrap as a prerequisite, it can be developed using another Nix installation running under an existing Linux kernel, assuming that the bootstrapped Nix will behave the same as the one on the development machine. Once both parts work individually, we can combine them to create the actual full-source bootstrap.

3.1 The Outer Bootstrap

As a basis for the outer bootstrap, we used *live-bootstrap*, which we introduced in Section 2.3. It provides us with a Linux OS as a starting point for bootstrapping Nix.

3.1.1 Setup

During development, we ran the bootstrap in a QEMU VM. Because we wanted to control the arguments passed to QEMU, and the Python script included with *live-bootstrap* was not flexible enough in that regard, we wrote a Makefile that creates disk images in *bare-metal* mode and starts a QEMU VM using them. We used almost the same setup as the provided script: KVM acceleration enabled, `kernel-irqchip=split` set as the machine type, and an `e1000` network interface controller (NIC). We diverged from the live-bootstrap setup by setting the CPU type to `host`, but this had no effect on the files produced by the bootstrap; all hashes still matched after introducing the change. Furthermore, since we were using *bare-metal* images, we had to use a display for I/O rather than the serial console. To be able to comfortably use QEMU over SSH without having to attach to the VM using a remote desktop viewer, we used the `-display curses` option, which makes QEMU render the screen into the terminal, as long as the software running in the VM has not switched the display out of text mode. That required a small change to the Linux kernel command line that the bootstrap uses when switching from *fiwix* to Linux, to prevent the kernel from switching into graphical mode. At a later point, we modified the generator script so that it does not start a VM in *QEMU*-Mode, allowing us to choose between the two modes using the `BOOTSTRAP_PLATFORM` variable of our Makefile.

After the main bootstrap chain finishes, the Makefile can be used to start a development VM that is a clone of the original bootstrap VM. It does this by leveraging the copy-on-write nature of the *QCOW2* disk image format we are using. That allows us to return to the state directly after the bootstrap without rerunning the entire bootstrap chain.

3.1.2 Extensions

Using our new Makefile-based tooling, we built what we are calling the *extension system*. An *extension* is an optional continuation of *live-bootstrap*'s bootstrap chain. It consists of a list of steps that can be executed after the original bootstrap chain finishes.

We tried to reuse as much of the existing tooling in *live-bootstrap* as possible. For this reason, extensions use the same syntax for defining steps and the bootstrap chain as *live-bootstrap*. The file layout of an extension follows that of the `steps` directory in the *live-bootstrap* repository: The order of steps is defined in the `manifest` file, each package is contained in a subdirectory, and the hashes of the build results are stored in the `SHA256SUMS.pkgs` file.

Using the same structure allows us to bundle one or more extensions into the bootstrap image, creating a single bootstrap chain that executes the *live-bootstrap* steps and then the extension, without requiring manual intervention. We realized this by extending the Python script that generates the disk images with a new `--extension` command-line option.

A significant part of the bootstrap chain runs on kernels that do not support physical address extensions and thus cannot use the entire system memory. In combination with an in-memory file system, this tightly limits the available storage capacity. As a result, bundling an extension that is too large leads to a crash. To mitigate that, the script packs the extension's `steps` directory into a compressed `.tar.xz` archive. The archive is included in the bootstrap disk image alongside a generated `improve` script that unpacks the steps. When appending the extension's manifest to the existing one, a step that runs the generated unpack script is placed before the extension's first step. The `SHA256SUMS.pkgs` files are concatenated without any special processing.

To bundle multiple extensions, the option can be used multiple times. The extensions are then executed in the order they were specified. When using the Makefile to run the bootstrap, the extensions to be bundled can be set via the `BOOTSTRAP_EXTS` variable.

Aside from being integrated into the initial bootstrap chain, extensions can also be applied to an existing *live-bootstrap* installation. For this purpose, the Makefile has a target that packs an extension into a disk image. By setting the `MOUNT_EXT` make option when starting a development VM, the specified extension can be packed into a disk image and attached to the VM, allowing us to test new changes to extensions quickly.

The extension is installed by mounting the disk image and starting the `run.sh` script on it. The script replaces the contents of the `/steps` directory, which remains on a finished *live-bootstrap* installation, with the extension's steps, runs the *script generator* to generate new bootstrap scripts from the extension's manifest, and then executes the first stage of the generated scripts (`/steps/1.sh`) using bash.

3.1.3 The `dev-setup` Extension

The first thing we created using this system was the `dev-setup` extension. Its purpose is not to be used as a part of the final bootstrap chain, but to verify that the extension system works as intended and to support development and testing of the `nix` extension we created after it.

It builds *busybox*, a collection of UNIX command line utilities [19], to fill in those which are missing on a regular *live-bootstrap* install (e.g., `clear`, `less`) and sets up *busybox*'s init system, allowing us to start background services with the OS. *busybox* also includes a minimal variant of the `vi` text editor, which we can use to edit the bootstrap files on the running system without having to reboot.

The service the init system is relevant for is the *dropbear* SSH server that the extension builds after *busybox*. With *dropbear* installed and running, we no longer need to rely on the serial console to interact with the VM and can instead connect to it directly over SSH.

To further improve the usability of the terminal interface, the extension configures the *bash* shell with a prompt that displays the username, the hostname, and the current working directory, as is common on other Linux distributions. Additionally, it installs scripts to simplify mounting and building an extension (`buildext`), and to build a single package directly (`build.sh`).

3.1.4 The `nix` Extension

Using the tools installed by the `dev-setup` extension, we developed an extension to bootstrap Nix. As a reference for writing the build scripts, we looked at how these packages are built in Arch Linux [20] and Alpine Linux [21]. We choose these distributions because of the similarity of their package managers' build scripts to the *live-bootstrap* ones and because Alpine Linux, like *live-bootstrap*, uses the *musl* C standard library.

When using the generator script to create the disk images, the file system does not fill the entire available space in the image. To fix this, the nix extension resizes the partition and file system to fill the entire available space before building any packages.

We decided to bootstrap version `2.28.4` of Nix because it is the version packaged in the Nixpkgs snapshot we will use for the inner bootstrap in Section 3.2.

Nix is written in C++. The needed compiler, `g++`, is already provided by *live-bootstrap*. Current versions of Nix—including the version we are using—are built using the *Meson* build system. *Meson* is written in Python, which, too, is already provided by *live-bootstrap*.

While the interpreter is present, building *Meson* also requires the Python packaging utilities (the packages `build`, `installer`, and `wheel`). The first build action of the extension is to bootstrap the necessary Python packages and to build *Meson*.

To execute a build, *Meson* requires the *Ninja* build system, which it uses in the background, to be installed. It is the package that the extension bootstraps next. Because *CMake* is required to build some of Nix's dependencies, the extension also bootstraps it.

Some libraries included with *live-bootstrap* are built for static linking only. To avoid rebuilding those libraries, we opted to link Nix and its dependencies statically as well. Among the libraries Nix depends on, there are a few that require special consideration:

- `bdw-gc`, `libgit2`, `nlohmann-json` and Nix itself need to have `-latomic` added to the `CFLAGS` environment variable to be built on 32-bit x86, because GCC does not automatically link against `libatomic`, which is used to implement the `__atomic` functions on platforms that do not support GCC's built-in hardware-based implementation. [22]

- We had to compile `libsodium`, a library providing cryptography functionality, with the stack protector disabled. While this is not optimal from a security standpoint, all packages we are building will be replaced with versions built from Nixpkgs, so we did not consider this a significant problem.

- Nix uses `queue.h`, a file that is present in *glibc*, but missing from *musl*. Alpine Linux provides an implementation of `queue.h` for *musl*, which Nixpkgs also uses. We download that file from the Alpine Linux repository and install it to the correct location so we can build Nix against *musl*. [4, `/pkgs/by-name/mu/musl/package.nix`]

- The version of `libarchive` included in *live-bootstrap* lacks `bzip2` support. Compiling against it results in a Nix that cannot unpack `.tar.bz2` archives. To overcome this, we needed to rebuild `libarchive`. Because the `bzip2` built by *live-bootstrap* lacks the *pkg-config* file needed for the *libarchive* build to detect it, we had to rebuild *bzip2* as well.

After building Nix, the extension's final step is to set it up. For this, we wrote an `improve` script that generates the `/etc/nix/nix.conf` configuration file, creates the `/nix` directory, and sets up the `nixbld` group and users, which Nix needs for its build sandbox.

Figure 1 shows the dependencies of the packages in the `nix` extension. For readability, we did not draw edges for dependencies that are already fulfilled transitively.

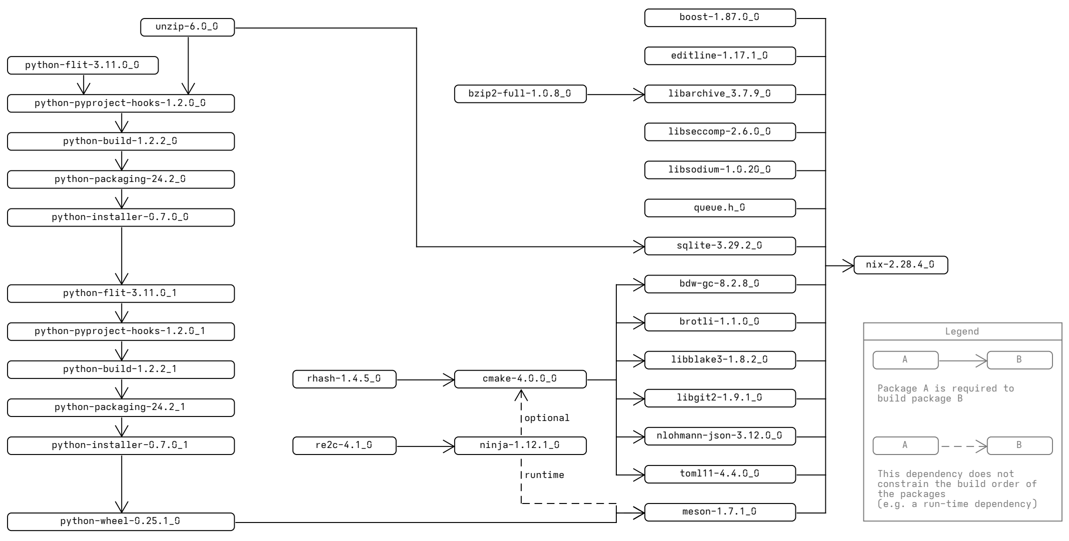

Figure 1: Dependency Graph of the `nix` extension

3.2 The Inner Bootstrap

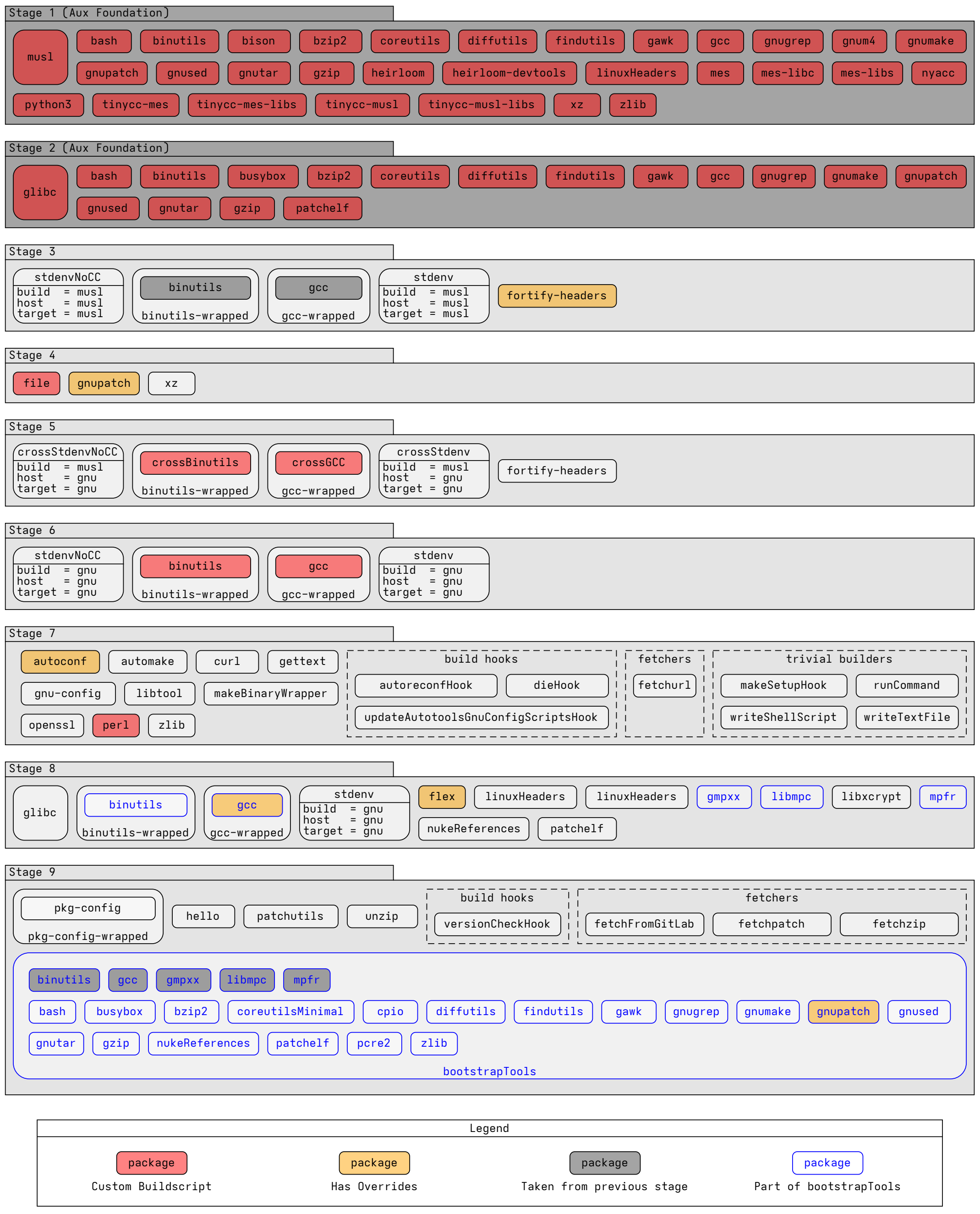

Figure 2: Overview of the packages built in each stage of the inner bootstrap

NixOS is a part of Nixpkgs. As such, building a full-source bootstrap chain for NixOS is equivalent to building one for Nixpkgs.

The approach we chose for the inner bootstrap was to focus on the `bootstrapTools` package from which every other package in Nixpkgs is bootstrapped. We used *Aux Foundation* as a starting package set to build a replacement for `bootstrapTools`.

Following the structure of *Aux Foundation*, we split the bootstrap chain into stages. Our final `bootstrapTools` bootstrap consists of nine stages. Each stage is a function that uses the derivations produced by the previous stages to produce a new attrset with derivations.

3.2.1 Stages 1 and 2: Aux Foundation

The first two stages re-export the *Aux Foundation* packages from the stages with the same names. It is worth noting that *Aux Foundation* has three stages, starting with stage 0, where it builds a first, simple C compiler. However, we did not use the packages from stage 0 directly. Therefore, our bootstrap chain starts with stage 1. The bulk of the packages we used are built in stage 2, including the `gcc` toolchain and the `glibc` C standard library. The `musl` C library against which the toolchain links is built in stage 1.

3.2.2 Stage 3: First Minimal `stdenv`

In stage 3, we build a first version of `stdenv`. In addition to a C toolchain, Nixpkgs' `stdenv` contains the following packages: [4, `/pkgs/stdenv/generic/common-path.nix`]

- `bash`
- `bzip2`
- `coreutils`
- `diffutils`
- `file`
- `findutils`
- `gawk`
- `gnugrep`
- `gnumake`
- `gnused`
- `gnutar`
- `gzip`
- `gnupatch`
- `xz`

As shown in Figure 2, *Aux Foundation* provides all of these packages, except for `file`, which we thusly omit in this stage.

To be usable in `stdenv`, the C compiler must be wrapped by a script that injects the command-line flags needed to produce working executables within the Nix build environment. Building this wrapper itself requires `stdenv`. Nixpkgs solves this by additionally building `stdenvNoCC`, a variant of `stdenv` that is built without a C compiler. We replicated this solution for our bootstrap.

Wrapping a compiler that links against `musl` additionally requires the `fortify-headers` package. To build it, we called its existing function in Nixpkgs with our package set.

The `patch` command built by *Aux Foundation* is broken when used as `stdenv` tries to invoke it, so we added an override to build the first version of `fortify-headers` without applying the patches. Downloading the sources requires `fetchurl`, which depends on `curl`. The `bootstrapTools` package does not include `curl`, meaning that Nixpkgs needs to solve this problem, too. To do so, Nixpkgs includes a bootstrap version of `fetchurl` that uses the `<nix/fetchurl.nix>` fetcher instead. We used this to provide the required `fetchurl`.

Notwithstanding Nixpkgs supporting building `bootstrapTools` using binaries linked against `musl`, we decided to use `glibc`-linked binaries like the prebuilt version that an unmodified Nixpkgs would use. The rationale was to replicate the original `bootstrapTools` as closely as possible to avoid causing problems when integrating the bootstrap chain into Nixpkgs. Since the `gcc` package provided by *Aux Foundation* is configured to link against `musl`, we cannot use this first `stdenv` to build the `bootstrapTools` replacement, and need to build a compiler that links against `glibc` first.

3.2.3 Stage 4: Complete `stdenv`

With the basic `stdenv` from the previous stage, we (re-)built `file`, `gnupatch`, and `xz` to complete the `stdenv` packages. While `file` is built for the first time in this stage, both `gnupatch` and `xz` are already contained in *Aux Foundation*. The packages work in isolation, but when used with `mkDerivation`, they are not able to apply patches and unpack `.tar.xz` archives respectively. We used the Nixpkgs package definitions for the rebuild to ensure the options required to make the programs work with `stdenv` are set correctly. Building `gnupatch`, however, again required an override to prevent it from attempting to apply patches, since the previous version is non-functional. To save on build time, we did not immediately rebuild `gnupatch` with the patches applied. The unpatched version is sufficient for building everything until it is rebuilt for inclusion in `bootstrapTools`.

3.2.4 Stage 5: Cross-Compiler

Using the now complete set of inputs, we built a new `stdenv`. With this, we rebuilt `fortify-headers` —this time without the override that removed the patches.

Until this point, every `stdenv` we built had `i686-unknown-linux-musl` set as its *build*, *host*, and *target* platforms. That means they used and produced binaries that can be executed on any i686-compatible (32-bit x86-based) CPU running Linux, and that the binaries are linked against `musl`.

The central operation of this stage is building a cross-compiler that is itself linked against `musl`, but outputs binaries linked against `glibc`. To achieve this, we rebuilt `binutils` and `gcc`, and their respective wrappers, with `i686-unknown-linux-gnu` set as the *target* platform. From these packages, we built a new `stdenv` variant that we called `crossStdenv`. This new `stdenv` has `i686-unknown-linux-musl` set as its *build* platform, and `i686-unknown-linux-gnu` as the *host* and *target* platforms. The `glibc` package we use in this stage is taken from *Aux Foundation*'s `stage2`.

3.2.5 Stage 6: Native Compiler

In the next stage, we used the `crossStdenv` to rebuild `binutils` and `gcc` again, this time for `i686-unknown-linux-gnu`, which they also target. For the build, we used the same platform definitions as `crossStdenv`. We created another `stdenv` that uses the native compiler, with `i686-unknown-linux-gnu` as its *build*, *host*, and *target* platforms.

3.2.6 Stage 7: `fetchurl`

To recreate the Nixpkgs-built `bootstrapTools` as closely as possible, we used the package definitions from Nixpkgs for all included packages. Because these were built with the expectation of having the entirety of Nixpkgs available, they use fetchers like `fetchpatch`, which depend on features of `fetchurl` that are not available in the bootstrap version. That is why, in this stage, we bootstrapped the regular `curl`-based `fetchurl`.

3.2.7 Stage 8: Final Compiler

As stated before, we wanted to build all packages contained in `bootstrapTools` from their Nixpkgs definitions. For this reason, we rebuilt `glibc`, `gcc`, and `binutils`, including their dependencies from their Nixpkgs definitions for the last time, and used the result to create our final `stdenv`.

3.2.8 Stage 9: `bootstrapTools`

Every package contained in `bootstrapTools` is either built for the first time or rebuilt in `stage8` or `stage9`. Doing this was necessary so that all packages in `bootstrapTools` are linked against the exact `glibc` included with them that we built in `stage7`. We obtained the `bootstrapTools` derivation by calling the function in `make-bootstrap-tools.nix`[16], and passing in our package set.

3.2.9 Integration

The `bootstrap-files`[17] directory in Nixpkgs contains a `.nix` file for every supported platform. Each file contains an attrset with two entries: `bootstrapTools` and `busybox`. These entries contain the results of two `<nix/fetchurl.nix>` calls that download the prebuilt `bootstrapTools` and a statically linked `busybox` binary, which is used as the initial builder. The files downloaded here are the ones `make-bootstrap-tools.nix` returns in the `bootstrapFiles` attribute. Because the attrset contained in that attribute has the same form as the ones defined in the `bootstrap-files` directory, we replaced the contents of `i686-unknown-linux-gnu.nix` with code that imports our bootstrap chain and returns the value of `bootstrapTools.bootstrapFiles` from it. For all other platforms, we modified the files to throw an error when evaluated, to prevent accidental use of non-bootstrapped packages.

Since January 2024, support for the `i686-linux` platform by Nixpkgs and thus NixOS has been discontinued. [23] Consequently, NixOS is no longer guaranteed to be buildable on this platform. Because we encountered build failures when building a NixOS `toplevel` derivation, we decided to build NixOS for the `x86_64-linux` platform instead. Thanks to 64-bit x86 CPUs generally being able to execute 32-bit x86 binaries, we can use our existing `i686-linux` bootstrap chain to cross-compile `bootstrapTools` for `x86_64-linux`: Based on the code in `make-bootstrap-tools-cross.nix`[18], we modified `x86_64-unknown-linux-gnu.nix` in the `bootstrap-files` directory to build the `x86_64-linux` version of `bootstrapTools` from the bootstrapped `i686-linux` Nixpkgs, making use of its cross-compilation support.

[16] [4, `/pkgs/stdenv/linux/make-bootstrap-tools.nix`]
[17] [4, `/pkgs/stdenv/linux/bootstrap-files/`]
[18] [4, `/pkgs/stdenv/linux/make-bootstrap-tools-cross.nix`]

3.3 The Combined Bootstrap

By linking the bootstrap chains together, we constructed a combined bootstrap chain that starts with *live-bootstrap* and results in a NixOS installation. We implemented this through another extension using the system we introduced in Section 3.1.2.

First, the `nixos` extension reconfigures the *tmpfs* mounted at `/tmp` to a size of 32 GiB and four million inodes. To support this without impacting the available system memory, it also generates and activates a swap file of the same size. Growing the *tmpfs* is necessary, as Nix stores the working directory of the build processes in there. The original size and inode count set by *live-bootstrap* caused space-intensive builds to run out of capacity. Unmounting the *tmpfs* entirely or instructing Nix to use another directory were solutions we considered as well. Both caused build failures in *Aux Foundation* —hence, we did not pursue them any further.

Before Nix is used to build any packages, we added a step that starts the *Nix daemon* and configures all subsequent Nix invocations to access the Nix store through the daemon rather than directly. Having this single point through which all interactions with the Nix store are managed prevented a race condition between multiple builds finishing at the same time, which had caused bootstrap failures before we introduced the daemon. The race condition might have been mitigated by deactivating the `auto-optimise-store` option in `/etc/nix/nix.conf`. If the option is enabled, Nix deduplicates all files in the Nix store using hard links. The build failure occurred when a Nix instance attempted to create a hard link that another build process had already created in the time since it checked for its presence. To save on disk space, we kept the feature enabled.

As a result of *live-bootstrap* targeting 32-bit x86, the kernel it builds can not execute 64-bit binaries, even when running on a 64-bit CPU. To overcome this limitation, we used Nix to cross-compile a 64-bit Linux kernel. Nixpkgs' default configuration is not suitable for use without an initial RAM file system (initramfs) containing the kernel modules needed to interface with the disk containing the root file system. We used the kernel configuration from *live-bootstrap*, ensuring that the new kernel is bootable on all (64-bit capable) computers that can run *live-bootstrap*. A `jump` script switches to the new kernel and resumes the bootstrap chain there.

During development of the kernel bootstrap, we found that the `bash` package from *Aux Foundations*'s `stage1` is not buildable under a 32-bit kernel. The cause is one of the checks in the package's `./configures` script crashing in this scenario. Before integrating the bootstrap chains, the problem did not occur because we developed the inner bootstrap on a NixOS installation with a 64-bit kernel. Our solution was to skip the misbehaving check and always assume that the functionality it was trying to detect is unavailable.[19]

Attempting to build 64-bit packages with the 32-bit Nix we built in Section 3.1.4 fails with a `Bad system call` error. We added a step that uses the 32-bit Nix to cross-compile a 64-bit version, as we did with the kernel. Afterward, the Nix daemon is restarted so that it, too, uses the 64-bit version of Nix.

In its penultimate step, the extension builds a NixOS `toplevel` derivation for the `x86_64-linux` platform. The configuration used to build it can not be adjusted for the specific machine the bootstrap is running on due to *live-bootstrap*'s reproducibility checks. Instead, we added a script to the configuration that runs NixOS's boot process. The script uses the `nixos-generate-config`[20] utility to automatically detect the needed options for the hardware, generating the `hardware-`

[19] Pull request with the fix we opened upstream: https://git.auxolotl.org/auxolotl/foundation/pulls/3

[20] [4, `/nixos/modules/installer/tools/nixos-generate-config.pl`]

`configuration.nix` file. This file is imported by the NixOS configuration, which the script rebuilds before rebooting. When executing the rebuild, the script removes itself from the configuration, so that it runs only once. In this step, we also create the `/etc/NIXOS_LUSTRATE` file, which tells NixOS to perform a cleanup of the system. Before the configuration is activated, everything that does not belong to the configuration or is listed in that file is moved to `/old-root`. [10, `#sec-installing-from-other-distro`]

Finally, the extension jumps into NixOS. We used the 64-bit kernel built with *live-bootstrap*'s config earlier instead of the one built with the NixOS `toplevel` derivation for this, because hardware detection only happens after the first boot, and therefore required modules may still be missing from the initramfs, rendering the NixOS kernel unbootable. We did, however, use the initramfs included in the `toplevel` derivation. Doing so ensures that `nixos-generate-config` can properly detect the boot device.

The bootstrap chain ends when the hardware-detection script included in the initial configuration reboots the system. If running inside QEMU using the command from our Makefile, the VM exits at this point instead of completing the reboot.

3.4 Taking the Bootstrap Offline

Checksums ensure the integrity of the files downloaded from the internet during the bootstrap. They are used for this purpose throughout the bootstrap chain. Nonetheless, we wanted the bootstrap to be executable without an internet connection to eliminate the possibility that any component downloads unverified data, thereby contaminating the bootstrapped system.

Through the use of the `--external-sources` option when generating the bootstrap files, *live-bootstrap* can already be used fully offline. The only problem we encountered was that it still requires a NIC to be connected to the bootstrap machine. Without that, the bootstrap fails when it tries to acquire an IP address via DHCP. We made the problematic step skippable by introducing a new `--offline` option. Furthermore, the `nix` extension can also be used offline without modification. It exclusively uses the `sources` files to obtain the packages' source code. Accordingly, they are handled by the `--external-sources` option, too.

Building the `nixos` extension offline, on the other hand, required a more sophisticated approach. The Nix packages utilize the *fetchers* to download their source files. Manually finding the URLs and hashes of all source files would not be feasible within a reasonable timeframe, due to the number of packages and their associated source files involved in building the `toplevel` derivation. Besides that, we needed to instruct Nix to use the local source files rather than attempting the download.

3.4.1 `fetchurl`

The `nix derivation show` command prints a JSON representation of a given store derivation. The output produced for the store derivation corresponding to the example derivation shown in Listing 3 in Section 2.1.2 is shown in Listing 5.

```json
{
  "/nix/store/3ab...9m4-example.drv": {
    "args": [ "-c", "echo hello world > $out" ],
    "builder": "/nix/store/aqb...qsy-bash-interactive-5.2p37/bin/bash",
    "env": {
      "builder": "/nix/store/aqb...qsy-bash-interactive-5.2p37/bin/bash",
      "name": "example",
      "out": "/nix/store/n1x...s2k-example",
      "system": "x86_64-linux"
    },
    "inputDrvs": {
      "/nix/store/5hh...p56-bash-interactive-5.2p37.drv": {
        "dynamicOutputs": {},
        "outputs": [ "out" ]
      }
    },
    "inputSrcs": [],
    "name": "example",
    "outputs": {
      "out": {
        "path": "/nix/store/n1x...s2k-example"
      }
    },
    "system": "x86_64-linux"
  }
}
```

Store paths have been shortened for readability.

Listing 5: JSON representation of an example store derivation

With the `--recursive` option, the generated JavaScript Object Notation (JSON) contains not only the contents of the store derivation passed to the command, but also of the store derivations of all store paths referenced by it, directly or indirectly. The result encompasses everything needed to build the input derivation from scratch.

FODs are trivial to detect in this representation, because their `outputs.out` attribute additionally contains the keys `hash`, `hashAlgo`, and `method`. The value of `method` determines how the hash is to be calculated: `flat` specifies that the derivation's output is a single file that should be hashed directly, whereas `nar` and `recursive` instruct Nix to calculate the hash from a NAR dump of the output.[21] [8, ch. 5.4.1.1] Entries created by a call to `fetchurl` also contain the URL they download in the `.env.url` attribute, or in the `env.urls` attribute if multiple mirrors are specified.

Initially, we focused only on files that *live-bootstrap*'s tooling can download. Every line in a `sources` file contains the URL, the SHA-256 hash, and a filename. To generate such a file, we wrote a script that parses the output of `nix derivation show --recursive` for all packages built in the `nixos` extension. It filters out everything that is not a FOD with a known URL, a `sha256` hash, the `flat` hashing method, and an unset or empty `.env.postFetch` value. The `postFetch` environment variable is used by fetchers that perform the actual download with `fetchurl`, but need to post-process the downloaded file. Nix checks the hash *after* the post-processing step, hence there is no way to infer the checksum of the original file.

To make Nix use the downloaded files, we added a new step to the `nixos` extension between starting the Nix daemon and building the first package, that copies the files into the Nix store with `nix-store --add-fixed`. This command results in the same store path as building a FOD with the file as its output would. [8, ch. 8.3.3.1] Nix thus detects that the store path is already present and skips the download. That is another reason why we had to filter out derivations with a non-empty `postFetch` value: Even if we were able to obtain the correct hash and download the files, in order to create a FOD with the correct store path, we would have had to replicate the post-processing step before adding the file to the Nix store.

3.4.2 Other Fixed Output Derivations

For all FODs we were unable to replace with this method, we used Nix's binary cache feature. We modified the script to copy all incompatible FODs into a local binary cache, which is packed into an archive and placed next to the downloaded source files. During the bootstrap, the archive is unpacked and the contained FODs are copied to the Nix store of the bootstrap machine.

The caveat with this solution is that post-processing no longer occurs on the bootstrap machine, but on the computer preparing the bootstrap, using non-trustworthy software. Given that FOD build results are checked against predefined hashes, we were willing to accept this tradeoff.

3.4.3 Built-in Fetchers

After implementing the binary cache, the last source files that were not available offline were those downloaded by the built-in fetchers. Since the built-in fetchers do not produce store derivations, there is nothing for `nix derivation show` to convert to JSON. Even though the downloaded store paths do appear in the attributes of the derivations that use them, paths obtained from a built-in fetcher do not have their own entries.

In Nixpkgs, built-in fetchers may not be used. [10, `#chap-pkgs-fetchers`] *Aux Foundation* does, however, use them in two places: First, to download a repository with library functions, and secondly, to download and unpack the sources for the `stage0` packages.

[21] There are further, experimental hashing methods. Those are however not relevant in the context of this thesis, as we did not enable the relevant experimental features.

We addressed the first instance by copying the contents of the library repository into the *Aux Foundation* source tree. That allowed us to remove the download entirely.

For the second instance, removing the calls to `builtins.fetchTarball` was not trivially[22] possible. Because these downloads occur before any Nix packages are built, the `fetchTarball` function is essential for unpacking the downloaded archives. Instead of removing the fetcher calls, we modified them so that the uniform resource locators (URLs) can be overwritten with `files://` URLs pointing to local copies of the files. This way, `fetchTarball` still unpacks the archives, but no longer requires an internet connection.

[22]There is an open pull request on the *Aux Foundation* repository [14, `#15`] that aims to replace the `fetchTarball` calls with a combination of the code behind `<nix/fetchurl.nix>` and the undocumented `builtin:unpack-channel` builder to solve this exact problem.

RESULTS 4

Previously, we described how we implemented the full-source bootstrap for NixOS. In this chapter, we present the final results we achieved using our implementation.

4.1 RQ1: Is it possible to construct a full-source bootstrap chain for the Nix package manager?

Using the approach detailed in Section 3.1, we were able to bootstrap a Linux OS with a fully operational Nix installation. In terms of files, the only prerequisites are the source code and a human-auditable binary seed; thus, we consider this a successful implementation of a full-source bootstrap chain for the Nix package manager.

4.2 RQ2: Is it possible to construct a full-source bootstrap chain for Nixpkgs by replacing the `bootstrapTools` package?

In Section 3.2, we showed that a replacement for the `bootstrapTools` package can be bootstrapped entirely using Nix packages. Because the packages are built from only source code and a human-auditable binary seed, we consider this, too, a full-source bootstrap chain.

4.3 RQ3: How can these two bootstrap chains be combined to create a full-source bootstrap for NixOS?

Using the solution we detailed in Section 3.3, NixOS can be bootstrapped from source. We extended the Nix bootstrap chain developed to answer **RQ1** to bootstrap a 64-bit Linux environment, switch to it, and then build and install a NixOS `toplevel` derivation from the modified Nixpkgs we developed to answer **RQ2**.

Because of availability issues with the source files, running the bootstrap only completed with manual intervention before we made it capable of running offline to answer **RQ4**. We successfully executed the entire bootstrap in a VM and on a physical computer. The VM was assigned 16 GiB of RAM and 12 logical CPU cores. It was executed using QEMU `9.2.4` on a host machine running Fedora 42, which was equipped with an *AMD Ryzen 7 5800X* processor and 32 GiB of RAM.

The exact QEMU command can be obtained from the Makefile located in the root directory of our fork of the *live-bootstrap* repository. The physical computer was a *Lenovo ThinkPad T440p* equipped with an *Intel Core i7-4702MQ* processor and 16 GiB of RAM.

In the VM, our three successful offline bootstrap runs took 17h 3min, 17h 21min, and 17h 43min, as measured by the `time` command. We did not measure the bootstrap duration on the physical computer.

4.4 RQ4: Can the bootstrap chain be executed fully offline?

With the adjustments we described in Section 3.4, we were able to run the bootstrap in a VM without any NIC attached. Consequently, this VM was not connected to the internet. These changes made the bootstrap independent of the availability of the servers hosting the source files, once they have been downloaded to the computer preparing the bootstrap.

After implementing the changes, the bootstrap in the VM passed without requiring any interaction. On the physical computer, the last step of the bootstrap chain failed without an internet connection because `nixos-generate-config` enabled options that required building additional packages. Furthermore, we had to remove the kernel arguments we added to be able to use QEMU's `curses` display mode from the configuration. With these set, the video output did not work after the bootstrap completed.

4.5 RQ5: Is it possible to create a NixOS ISO image that can be used to install the bootstrapped NixOS on other computers without having to run the full bootstrap on them first?

We successfully built a NixOS installer ISO image from our modified version of Nixpkgs by following the instructions in the NixOS manual [12, `#sec-building-image`] on a NixOS installation that resulted from running the full-source bootstrap.

Using this image, we were able to install NixOS on another VM without running the entire bootstrap chain. Since the installer ISO only contains the run-time dependencies of the packages it includes, using it to install NixOS still involves building through the inner part of the bootstrap chain. Because of the source availability issues we encountered, the installation only succeeded after we loaded the source files we downloaded for running the bootstrap offline into the Nix store of the installation system.

4.6 RQ6: How can the bootstrap be adapted to architectures beyond x86-based systems?

Neither *live-bootstrap* nor *Aux Foundation* support architectures other than 32-bit x86 at the time of writing. That means that it is not possible to execute the bootstrap chain on other architectures directly. With the Nix DSL code shown in Listing 6, we were able to cross-compile a NixOS installer ISO image for the `aarch64-linux` platform that we confirmed to be bootable on a Raspberry Pi 4 with UEFI firmware [24] installed. Nonetheless, because we did not implement a `bootstrapTools` bootstrap for that platform, it was not possible to build packages on the booted installation system. Cross-compiling `bootstrapTools` from `i686-linux` like we did for `x86_64-linux` would be a solution, but would require a secondary x86-based computer set up as a remote builder, [8, ch. 7.2] to run the `i686-linux` parts of the bootstrap chain on. Our ability to cross-compile the installer ISO for `aarch64-linux`, however, demonstrates that it would be possible to use an x86-based computer to cross-compile the entire system configuration for other platforms.

cross-iso.nix Nix

```nix
(import <nixpkgs/nixos> {
  configuration = builtins.toFile "configuration.nix" ''
    { ... }:
    {
      imports = [ <nixpkgs/nixos/modules/installer/cd-dvd/installation-cd-
      minimal.nix> ];

      nixpkgs.buildPlatform = "x86_64-linux";
      nixpkgs.hostPlatform = "aarch64-linux";
    }
  '';
}).config.system.build.isoImage
```

Listing 6: Nix DSL expression for cross compiling an `aarch64-linux` installer ISO on `x86_64-linux`

4.7 RQ7: Does the bootstrap chain still include any binaries besides the initial bootstrap seeds?

Without employing any complex methods, we can confirm this by providing an example: the basic NixOS configuration we used to build the `toplevel` derivation for the NixOS bootstrap depends on `rustc`, the compiler for the *Rust* programming language. That compiler is bootstrapped from precompiled binaries provided by the *Rust* project. [4, `pkgs/development/compilers/rust/bootstrap.nix`]

5 DISCUSSION

Having demonstrated that and how a full-source bootstrap for NixOS can be implemented, we will now delve into the broader implications for the security and usability of the bootstrapped OS. Moreover, we will discuss the challenges we faced during implementation regarding the availability of the source code.

5.1 Usability

Being built from the same sources, the bootstrapped NixOS installation behaves just like a non-bootstrapped one from the end-user perspective. The differences lie in the initial installation procedure and in the process of installing additional packages.

5.1.1 Initial Installation

Our introduction of the full-source bootstrap significantly increases the computational cost of installing NixOS—especially when running the entire bootstrap chain starting with *live-bootstrap*. Moreover, the bootstrap, at least in its current form, limits hardware compatibility to just `x86_64-linux`: The bootstrap chains we built upon only run on x86 CPUs, and NixOS only supports 64-bit x86. Additionally, *live-bootstrap* limits us to computers that can boot from a master boot record (MBR), excluding machines that only support the newer UEFI standard.

The requirement for MBR compatibility, as well as parts of the computational cost, can be worked around by using the NixOS installer image. As we showed, the installer image can be built from our modified Nixpkgs on a bootstrapped NixOS installation. Including the build-time dependencies needed for a basic NixOS system might be a viable approach to reduce the installation's resource requirements further.

Using an installation medium merely shifts the problem, though: Obtaining said installation media without breaking the trustworthiness of the binaries established through the bootstrap still requires building it oneself. Still, this may present an improvement over having to run the entire bootstrap chain for every installation when setting up multiple computers, or when a trusted party provides the once-built image to multiple others. We will discuss another possible solution to this issue in Section 5.4.

5.1.2 Adding Packages

As long as the build-time dependencies compiled during the initial installation remain in the Nix store, installing additional packages has a similar computational cost to using another source-based package manager. When the garbage collector is triggered for the first time, however, the build-time dependencies are deleted. Afterward, installing new packages additionally involved rebuilding their build-time dependencies. In a test we performed[23], this included the entire inner bootstrap chain.

A way to improve this might be to explicitly include all, or a central subset of the build-time dependencies in the system configuration, to prevent them from being deleted during garbage collection. That might not be desirable in all scenarios, though, as it limits the amount of disk space the garbage collector can free: On a fresh install, garbage collection freed more than 19 GiB, which would keep being used if deleting the build-time dependencies was prevented.

5.1.3 Updating

The issue of updating is twofold: the modified version of Nixpkgs must be updated with package definitions from upstream Nixpkgs, and the bootstrapped NixOS installations must be updated with the updated Nixpkgs.

Since we implemented the inner bootstrap in a fork of the Nixpkgs repository, updating the Nixpkgs version it bootstraps requires manually rebasing our changes onto a newer Nixpkgs commit using Git. Being a manual process, as well as the potential for conflicting changes in the upstream repository that we would have to reconcile manually, render this setup unsuitable as a long-term solution.

Ideally, the inner bootstrap will be merged into the upstream Nixpkgs repository, eliminating the need for a modified fork. That would allow users to weigh usability against the trustworthiness of binaries by choosing whether to use the binary cache, use the prebuilt installation media, or build everything themselves.

Another option worth investigating is converting the bootstrap into a Nixpkgs overlay. Overlays are Nix language functions that override parts of the package set. As an overlay, the bootstrap would no longer be bound to any specific Nixpkgs version.

Updating the bootstrapped NixOS installations incurs the same computational cost as installing new packages. Additionally, any dependents of the updated packages must be rebuilt, too. Updates to `stdenv` or any of its dependencies consequently require a rebuild of all packages installed on the system.

5.2 Source Availability

A persistent problem we faced during implementation was the availability of the source code of the packages we needed to build. J. Malka et al. [25] found that 99.94% of the packages in a Nixpkgs snapshot from 2017 could still be built when they conducted their experiment six years later. From our experience, it is evident that they achieved this result only because they used the Nix project's binary cache—an outcome they anticipated in their paper.

[23]Building `stdenv` after running `nix-collect-garbage -d` on a bootstrapped NixOS installation using a default configuration generated with `nixos-generate-config`

For our bootstrap, we deactivated the binary cache to rule out downloading any precompiled binaries from it. Due to the aforementioned availability issues, we were only able to run the online version of the bootstrap with manual intervention, prompting us to focus our work on the offline version.

The two classes of build failures we experienced with FODs were files not being available at their URLs, either temporarily or permanently, and the hash of the source files available at their original URLs changing. The latter occurred with files generated on demand by Git repository hosting software. Presumably, the hashes changed due to changes to said hosting software. For a subset of those FODs, we addressed the changed hashes by cherry-picking the commits that updated the hashes from the upstream Nixpkgs repository into our own fork. We obtained the remaining unavailable files by selectively pulling the FODs from the binary cache.

While the binary cache solves the source availability problem in the context of Nix, it highlights the value of projects like *Software Heritage* [26], which archive source code repositories to keep them available even when the originals disappear from the internet.

5.3 Remaining Sources of Untrustworthiness

The full-source bootstrap we implemented effectively mitigates the attack via malicious compiler, as described by Thompson, with one notable exception: Compilers that are not bootstrapped from the C-compiler contained in `bootstrapTools`, but from a different, precompiled binary seed. Eliminating these other binary seeds from Nixpkgs remains an obstacle on the path towards establishing the ultimate trust that all packages produced by Nixpkgs' build definitions accurately reflect the source code they were built from.

5.3.1 External Factors

Despite the bootstrap chain's binary seed being small enough to be human-auditable, the bootstrap cannot be considered fully independent of any intransparent binaries: it still requires an existing OS to prepare the bootstrap files, as well as the firmware powering the hardware on which the bootstrap runs.

While it is conceivable for the OS used for preparation to be malicious and to tamper with the files as the bootstrap media is prepared, malicious code being present in the computer's firmware is the more realistic scenario. This attack vector does not require tampering with the files or the bootstrapped operating system itself, and cannot be prevented on the software side.

5.3.2 Malicious Source Code

Even when we trust the compiler to translate source code to machine code faithfully, we can not automatically trust the compiled binaries to be free of trojan horses: The attack on `xz-utils` [27], a data compression program and library, has made it abundantly clear, that being a widely used[24] FOSS project does not mean that a program does not include malicious code.

In this incident, an attacker managed to introduce a trojan horse targeting the *OpenSSH* [28] server. It allowed bypassing the SSH server's authentication check, granting the attacker full remote access to compromised systems. The malicious code was obfuscated well enough that it was not

[24]Some examples: Part of Nixpkgs' `stdenv`; Compression method used for NARs in Nix binary caches; Used by some projects to compress their source tarballs; Previously used to compress Arch Linux packages

found upon its introduction into the publicly available `xz-utils` source code repository. It was only discovered when a developer detected anomalous behavior of the SSH server on an affected machine. [29]

5.4 Reproducible Builds

Reproducible builds offer promising solutions to the computational cost problem we discussed in Section 5.1. A build is considered reproducible if, and only if, executing it with the same inputs produces a bit-by-bit identical result every time. That allows using prebuilt binaries without having to trust any single distributor:[25] if a quorum of independent builders produce the same output for a given build job, we can be reasonably sure that the build result has not been tampered with. [30]

J. Malka et al. [31] found that in the Nixpkgs revisions they investigated, between 69% and 91% of packages could be built reproducibly. The minimal installer image includes a higher share of reproducible packages: over 95% for all examined revisions since May 2019. In 2023, a version of the minimal installer image was successfully reproduced. [32], [33]

With a fully reproducible installer image, running the outer bootstrap can be skipped in favor of installing via the image, without sacrificing any of the trustworthiness improvements provided by the full-source bootstrap. The same is true for any other reproducible package individually. Even at the worst case observed by J. Malka et al. of 69% of packages being buildable reproducibly, a binary cache containing only independently verified binaries of these packages could significantly speed up the installation of the OS, and of individual packages. If most packages were available from an independently verified source, users might be motivated to compile the missing packages locally. An implementation of such a binary cache that verifies the packages it serves by comparing the build results of multiple builders is *Trustix*. [34], [35]

5.4.1 Content-Addressed Derivations

Nix can operate under two different models: The *extensional model* [5, pp. 87–134], which current versions of Nix use by default, and the *intensional model* [5, pp. 135–163], the implementation of which is still considered experimental.

In Section 2.1.1, we explained that the hash portion of a derivation's store path is calculated from its inputs. The derivations for which this is the case are called *input-addressed derivations*. [8, ch. 4.4.1] We also introduced an exception from this rule in the form of *fixed-output derivations (FODs)*. Both *input-addressed* and *fixed-output* derivations are part of the *extensional model*.

Under the *intensional model*, input-addressed derivations are replaced by content-addressed derivations (ca-derivations). As the name suggests, ca-derivations do not calculate their hash from their inputs, but from the contents of the resulting store path. They share this property with FODs, which can be considered a special case of ca-derivations. Unlike for FODs, the store path of a ca-derivation is not known at evaluation time. Hence, the advantage of FODs—that the build only has to be executed once if multiple derivations yield the same result[26]—does not apply to ca-derivations.

Using ca-derivations allows implementing an *early-cutoff* for the build process: When a dependency of a package needs to be rebuilt, this ordinarily means that its output path changes, which in turn

[25] Or in the nix context, the operators of a binary cache.

[26] e.g. if multiple `fetchurl` calls with different URLs but the same hash exist, or if the file has already been added to the store with `nix-store --add-fixed`.

changes the inputs of the dependent package, prompting it to be rebuilt as well. If the dependency is a ca-derivation and its rebuild produces a bit-by-bit identical result, it will produce the same store path, thus leaving the dependent package's inputs unchanged and allowing its rebuild to be skipped. [36], [37]

The *early-cutoff* can help avoid unnecessary rebuilds of large parts of the package set during updates: Suppose an update to a build tool fixes a rare edge case. With input-addressed derivations, this change requires rebuilding every package that depends on the tool directly or indirectly. When using ca-derivations, on the other hand, only the tool's direct dependents are rebuilt.[27]

[27] As long as none of them trigger the hypothetical, fixed edge case, leading to a change in the build result.

6 RELATED WORK

6.1 GNU Guix

GNU Guix [38] is another implementation of the “purely functional software deployment model” devised by Dolstra [5] as the theoretical foundation of Nix. It is itself based on Nix, but replaces the Nix Language with *Guile Scheme*, a *Lisp* dialect, as the language package definitions are written in. Like the Nix project, *GNU Guix* includes a repository with build definitions. [38, `/gnu`] The *Guix System* is a Linux distribution built from those packages and fills the role of NixOS in the *Guix* ecosystem. In 2023, members of the project announced that they successfully reduced the bootstrap seed of the *GNU Guix* package set to a 357-byte binary. For this, they coined the term “full-source bootstrap”. They claim to be the first distribution to achieve this. [3]

6.2 Nixpkgs’ `minimal-bootstrap`

When the full-source bootstrap for *GNU Guix* was published, efforts were made to implement a similar bootstrap chain for Nixpkgs. That resulted in the `minimal-bootstrap` package set. [4, `/pkgs/os-specific/linux/minimal-bootstrap`]

Neither the `minimal-bootstrap` package set, nor the open pull request (PR) that would extend it with additional packages [4, `#260193`] have been updated since 2023. In the Nixpkgs snapshot we used for our bootstrap, the `minimal-bootstrap` packages no longer built successfully. *Aux Foundation*, which we used instead, is based on the work on the `minimal-bootstrap` package set. It already incorporates the packages the PR would add. More importantly, it is currently under active development, and we were able to build the contained packages without modifications.

6.3 Other Operating Systems

Outside the realm of functional package managers, there are other operating systems that can be built using another existing OS’s tooling. Likely, a full-source bootstrap for any OS buildable on a different Linux distribution can be achieved using *live-bootstrap*.

[28]Gentoo can be installed in a subdirectory on another Linux distribution. [41] It might be possible to build installation media in this environment.

Examples include *FreeBSD* [39, ch. 26.9], *NetBSD* [40, ch. 33] and Gentoo[28], as well as Linux distributions that are designed to be always built on an existing Linux host system, like *Linux From Scratch* [42] and the embedded distributions *Yocto* [43] and *buildroot* [44].

6.4 Reproducible Builds

As we have shown in Section 5.4, bootstrapping and reproducibility are closely linked. The *Reproducible Builds* project [45] advocates for developers and distributors to take reproducibility into account when writing and packaging software. Under the project's umbrella, tools for making builds reproducible and for investigating non-deterministic behavior are developed.

CONCLUSION 7

In this thesis, we showed how the NixOS Linux distribution can be bootstrapped (almost) entirely from source code, eliminating the risk of the “trusting-trust” attack outlined by Thompson. [2] We implemented a bootstrap chain for the Nix package manager based on *live-bootstrap* and one for the Nixpkgs package set based on *Aux Foundation*. From these two bootstrap chains, we constructed a single, continuous bootstrap chain for NixOS.

To run the bootstrap offline, we added functionality to download the required source files once, before the bootstrap is executed. Because we consistently experienced problems with the availability of source files, this was a required step to complete the bootstrap without manual intervention. With these changes, we were able to run the entire bootstrap chain both in a VM and on physical hardware.

Additionally, we showed that the bootstrapped OS can be used to create installation media for other computers. That is also possible for foreign architectures through cross-compilation.

While it increases the trustworthiness of the resulting OS, running the entire full-source bootstrap increases the installation procedure’s computational cost drastically. We discussed how binary caching and installation media can alleviate this and offer users flexibility in how much time and resources they are willing to invest in the security of their OS. Extending on that, we explained how reproducible builds may make this consideration redundant by allowing build results to be verified through consensus among independent builders.

7.1 Future Work

To make it available to all Nix and NixOS users, we propose integrating the inner bootstrap chain into upstream Nixpkgs. For this purpose, the packages introduced by *Aux Foundation* and our bootstrap should be added to the existing `minimal-bootstrap` package set. To avoid breaking compatibility with non-x86 platforms, the full-source bootstrap should be introduced only for platforms where the required bootstrap seeds are available. The remaining platforms continue to use the precompiled `bootstrapTools` packages.

The bootstrap chain we built is not necessarily optimal yet. It might be possible to speed up the build process by removing superfluous (re-)builds. For example, it might be possible to build the 64-bit bootstrap tools entirely with a `musl`-linked 32-bit toolchain, eliminating the need to build a `glibc`-linked toolchain first. It would likely be even quicker to build the 64-bit bootstrap tools directly, without using an `i686-linux` Nixpkgs instance for cross-compilation.

Another area where additional work is needed is investigating which other precompiled bootstrap compilers Nixpkgs uses and how they can be replaced with bootstrapped versions. Ideally, as long as proprietary packages are not enabled, every installable package in Nixpkgs should be compiled from source, either locally or by a builder that pushes it to a binary cache.

LIST OF ABBREVIATIONS

CPU	central processing unit
DHCP	the Dynamic Host Configuration Protocol
DSL	domain-specific language
FHS	the Filesystem Hierarchy Standard
FOD	fixed-output derivation
FOSS	free and open-source software
GCC	the GNU Compiler Collection
GRUB	the Grand Unified Bootloader
GiB	Gibibyte(s)
HTTP	the Hypertext Transfer Protocol
I/O	input/output
IP	Internet Protocol
JSON	JavaScript Object Notation
KVM	Kernel-based Virtual Machine
MBR	master boot record
NAR	Nix Archive
NIC	network interface controller
OS	operating system
PR	pull request
QCOW	QEMU Copy-On-Write
RAM	random access memory
RPM	the RPM Package Manager
SHA-256	the Secure Hash Algorithm (256 bit)
SSH	secure shell
UEFI	the Unified Extensible Firmware Interface
URL	uniform resource locator
VM	virtual machine
attrset	attribute set
ca-derivation	content-addressed derivation
dpkg	the Debian Package Manager
initramfs	initial RAM file system
tcc	the Tiny C Compiler

LIST OF FIGURES

LIST OF SOURCE CODE

REFERENCES

[1] “Portage.” Accessed: Sep. 29, 2025. [Online]. Available: https://gitweb.gentoo.org/proj/portage.git/

[2] K. Thompson, “Reflections on trusting trust,” *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, Aug. 1984, doi: 10.1145/358198.358210.

[3] J. Nieuwenhuizen and L. Courtès, “The Full-Source Bootstrap: Building from source all the way down,” Apr. 26, 2023. Accessed: Sep. 25, 2025. [Online]. Available: https://guix.gnu.org/en/blog/2023/the-full-source-bootstrap-building-from-source-all-the-way-down/

[4] “Nixpkgs.” Accessed: Jul. 21, 2025. [Online]. Available: https://github.com/NixOS/nixpkgs/tree/92c2e04a475523e723c67ef872d8037379073681

[5] E. Dolstra, “The purely functional software deployment model,” Utrecht University, Netherlands, 2006. [Online]. Available: http://dspace.library.uu.nl/handle/1874/7540

[6] A. Hemel, “NixOS: the Nix based operating system,” Zenodo, 2006. doi: 10.5281/zenodo.12906987.

[7] LSB Workgroup, “Filesystem Hierarchy Standard,” Mar. 2015. Accessed: Oct. 03, 2025. [Online]. Available: https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html

[8] The Nix documentation team, “Nix Reference Manual.” Accessed: Sep. 25, 2025. [Online]. Available: https://nix.dev/manual/nix/2.28/introduction

[9] D. Marakasov, “Repository size/freshness map.” Accessed: Oct. 02, 2025. [Online]. Available: https://repology.org/repositories/graphs

[10] The Nix documentation team, “Nixpkgs Reference Manual.” Accessed: Oct. 02, 2025. [Online]. Available: https://nixos.org/manual/nixpkgs/stable

[11] “Nix.” Accessed: Jul. 24, 2025. [Online]. Available: https://github.com/NixOS/nix/tree/2.28.4

[12] The Nix documentation team, “NixOS Manual.” Accessed: Oct. 03, 2025. [Online]. Available: https://nixos.org/manual/nixos/stable

[13] “Aux.” Accessed: Oct. 03, 2025. [Online]. Available: https://auxolotl.org/en/

[14] J. Hamilton and others, “Aux Foundation.” Accessed: Sep. 25, 2025. [Online]. Available: https://git.auxolotl.org/auxolotl/foundation

[15] J. Orians, “bootstrap-seeds.” Accessed: Oct. 03, 2025. [Online]. Available: https://github.com/oriansj/bootstrap-seeds/tree/b1263ff14a17835f4d12539226208c426ced4fba

[16] S. Tyler and others, "live-bootstrap." Accessed: Sep. 25, 2025. [Online]. Available: https://github.com/fosslinux/live-bootstrap

[17] R. Masters, "builder-hex0." Accessed: Oct. 03, 2025. [Online]. Available: https://github.com/ironmeld/builder-hex0/tree/a2781242d19e6be891b453d8fa827137ab5db31a

[18] "Fiwix." Accessed: Oct. 03, 2025. [Online]. Available: https://www.fiwix.org/

[19] D. Vlasenko and others, "BusyBox." Accessed: Oct. 09, 2025. [Online]. Available: https://busybox.net/

[20] "Arch Linux Packages." Accessed: Oct. 14, 2025. [Online]. Available: https://gitlab.archlinux.org/archlinux/packaging/packages

[21] "Alpine Linux aports repository." Accessed: Oct. 14, 2025. [Online]. Available: https://gitlab.alpinelinux.org/alpine/aports

[22] A. MacLeod, "External Library Interface." Accessed: Oct. 14, 2025. [Online]. Available: https://gcc.gnu.org/wiki/Atomic/GCCMM/LIbrary

[23] R. Lahfa, "Limited cache availability for i686 (32 bits x86) architecture." Accessed: Oct. 14, 2025. [Online]. Available: https://discourse.nixos.org/t/limited-cache-availability-for-i686-32-bits-x86-architecture/37626

[24] Pi Firmware Task Force, "Raspberry Pi 4 UEFI Firmware Images." Accessed: Oct. 18, 2025. [Online]. Available: https://github.com/pftf/RPi4

[25] J. Malka *et al.*, "Reproducibility of Build Environments through Space and Time," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, in ICSE-NIER'24. Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 97–101. doi: 10.1145/3639476.3639767.

[26] "Software Heritage." Accessed: Oct. 21, 2025. [Online]. Available: https://www.softwareheritage.org/

[27] "CVE-2024-3094." Accessed: Oct. 20, 2025. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2024-3094

[28] The OpenBSD Project, "OpenSSH." Accessed: Oct. 21, 2025. [Online]. Available: https://www.openssh.com/

[29] P. Przymus and T. Durieux, "Wolves in the Repository: A Software Engineering Analysis of the XZ Utils Supply Chain Attack," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, 2025, pp. 91–102. doi: 10.1109/MSR66628.2025.00026.

[30] C. Lamb and S. Zacchiroli, "Reproducible Builds: Increasing the Integrity of Software Supply Chains," *IEEE Software*, vol. 39, no. 2, pp. 62–70, Mar. 2022, doi: 10.1109/MS.2021.3073045.

[31] J. Malka *et al.*, "Does Functional Package Management Enable Reproducible Builds at Scale? Yes.," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, Apr. 2025, pp. 775–787. doi: 10.1109/MSR66628.2025.00115.

[32] "Reproducible Builds in October 2023," Oct. 2023. Accessed: Oct. 22, 2025. [Online]. Available: https://reproducible-builds.org/reports/2023-10

[33] A. Engelen, "NixOS Reproducible Builds: minimal installation ISO successfully independently rebuilt." Accessed: Oct. 22, 2025. [Online]. Available: https://discourse.nixos.org/t/nixos-reproducible-builds-minimal-installation-iso-successfully-independently-rebuilt/34756

[34] A. Hoese, “Trustix: Distributed trust and reproducibility tracking for binary caches.” Accessed: Oct. 22, 2025. [Online]. Available: https://www.tweag.io/blog/2020-12-16-trustix-announcement/

[35] “Trustix.” Accessed: Oct. 22, 2025. [Online]. Available: https://github.com/nix-community/trustix

[36] T. Hufschmitt, “Implementing a content-addressed Nix.” Accessed: Oct. 21, 2025. [Online]. Available: https://www.tweag.io/blog/2021-12-02-nix-cas-4/

[37] T. Hufschmitt, “RFC 0062: Content-addressed paths.” Accessed: Oct. 21, 2025. [Online]. Available: https://github.com/tweag/rfcs/blob/cas-rfc/rfcs/0062-content-addressed-paths.md

[38] “Guix.” Accessed: Sep. 25, 2025. [Online]. Available: https://codeberg.org/guix/guix

[39] The FreeBSD Documentation Project, “FreeBSD Handbook.” Accessed: Oct. 22, 2025. [Online]. Available: https://docs.freebsd.org/en/books/handbook/book

[40] The NetBSD Developers, “The NetBSD Guide.” Jan. 01, 2025. Accessed: Oct. 22, 2025. [Online]. Available: https://www.netbsd.org/docs/guide/en/netbsd.html

[41] “Gentoo Prefix.” Accessed: Oct. 22, 2025. [Online]. Available: https://wiki.gentoo.org/wiki/Project:Prefix

[42] G. Beekmans, *Linux From Scratch*, 12.4th ed. 2025. Accessed: Oct. 22, 2025. [Online]. Available: https://www.linuxfromscratch.org/lfs/downloads/stable/LFS-BOOK-12.4.pdf

[43] The Linux Foundation, “Yocto Project Quick Build.” Accessed: Oct. 22, 2025. [Online]. Available: https://docs.yoctoproject.org/brief-yoctoprojectqs/index.html

[44] “The Buildroot user manual.” Accessed: Oct. 22, 2025. [Online]. Available: https://buildroot.org/downloads/manual/manual.html

[45] “Reproducible Builds.” Accessed: Oct. 22, 2025. [Online]. Available: https://reproducible-builds.org/