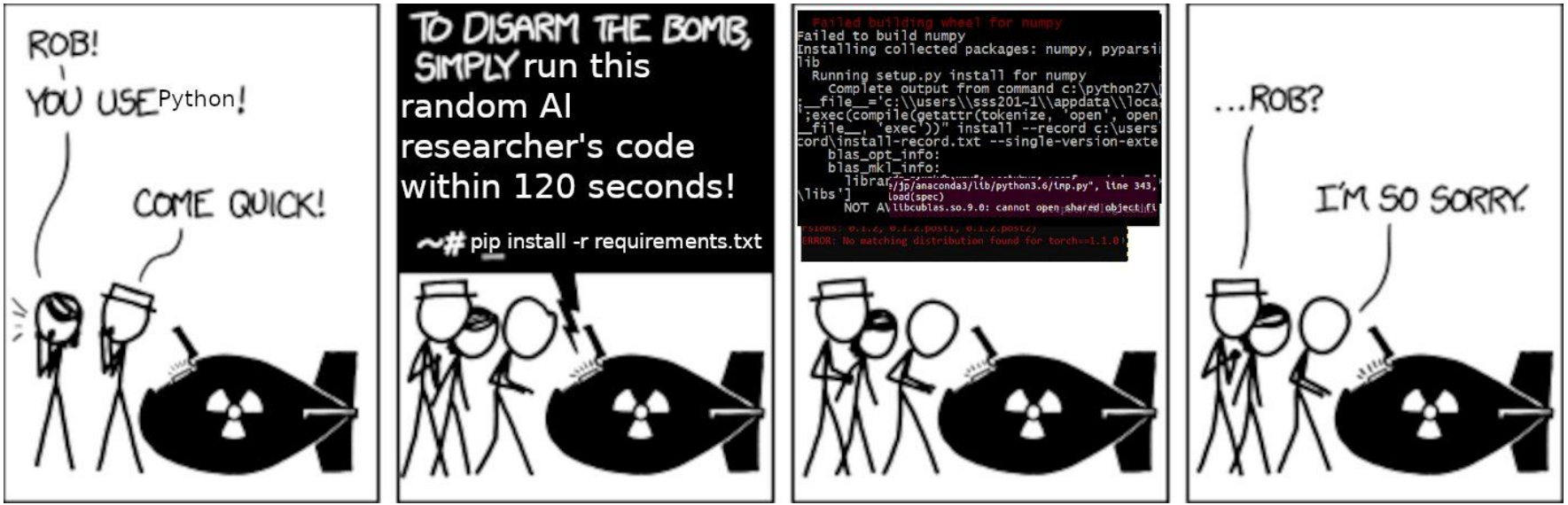
BuildXYZ

Automatic dispenser of native dependencies

Ryan Lahfa
Advisor: Jörg Thalheim
Chair of Distributed Systems and Operating Systems
https://dse.in.tum.de/

01.03.2023 – 31.07.2023

Motivation : state of affairs



Challenge: existing build systems and package managers have too many *implicit* dependencies.

Motivation : state of affairs

Table 3: Characterization of package dependency graphs (without disconnected nodes)

	npm	PyPI
#Nodes	577943	84188
Avg node outdegree	4.27	2.95
Avg dependency tree size	86.55	7.33
Avg dependency tree depth	4.39	1.71

Challenge (supply chain) : existing build systems and package managers have **too many** dependencies.

Motivation : state of affairs

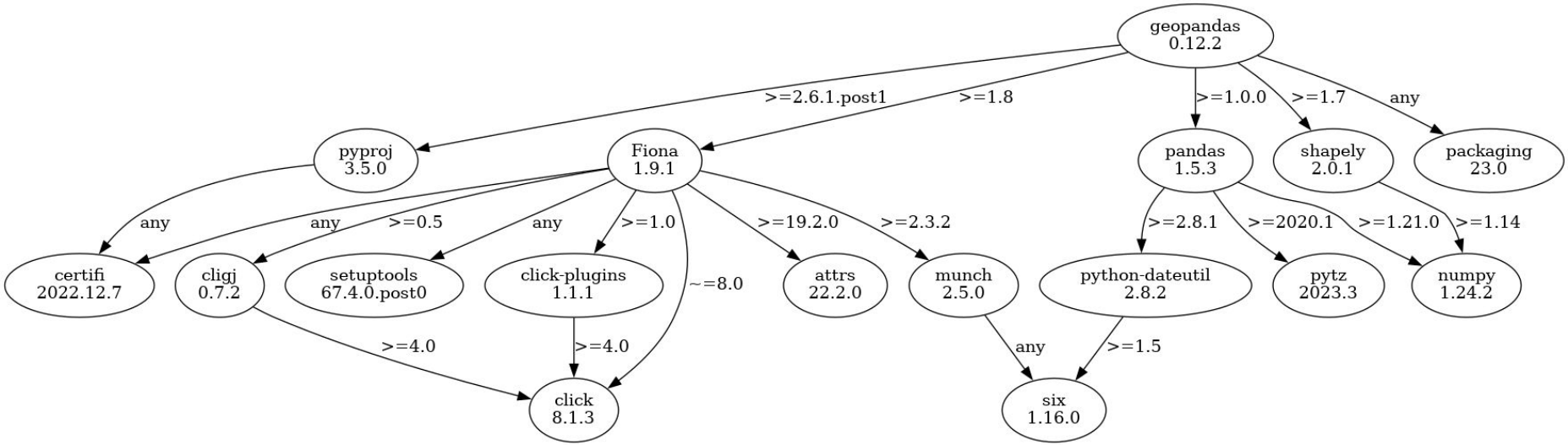
Bundled Wheels on Linux

While we acknowledge many approaches for dealing with third-party library dependencies within `manylinux1` wheels, we recognize that the `manylinux1` policy encourages bundling external dependencies, a practice which runs counter to the package management policies of many linux distributions' system package managers [14], [15]. The primary purpose of this is cross-distro compatibility. Furthermore, `manylinux1` wheels on PyPI occupy a different niche than the Python packages available through the system package manager.

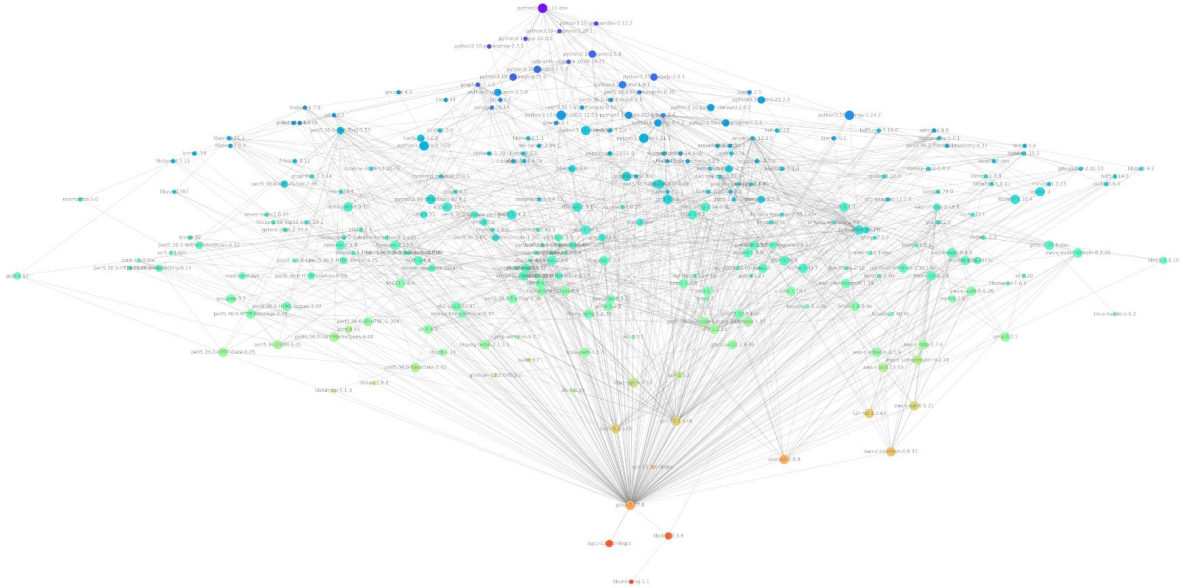
`manylinux1` wheels distributed through PyPI that bundle security-critical libraries like OpenSSL will thus assume responsibility for prompt updates in response disclosed vulnerabilities and patches. This closely parallels the security implications of the distribution of binary wheels on Windows that, because the platform lacks a system package manager, generally bundle their dependencies. In particular, because it lacks a stable ABI, OpenSSL cannot be included in the `manylinux1` profile.

Challenge (supply chain) : existing build systems and package managers are **walking security hazards.**

Motivation : a Python dependency graph



Motivation : the realization on a NixOS system

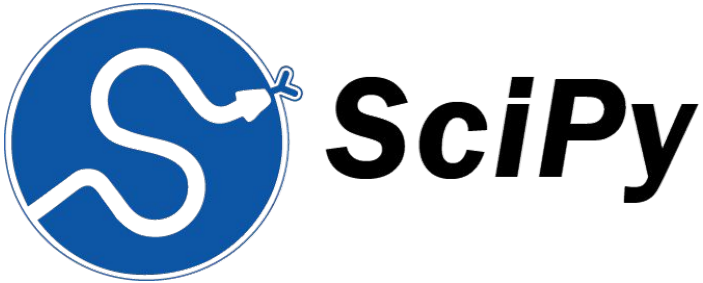


Challenge : existing build systems and package managers relies on **many native system libraries.**

A word on the impact for research **itself**

Python packaging ecosystem has been instrumental in the proliferation of Python packages for a wide range of scientific applications.

- Python at CERN: ROOT framework for PB-scale data analysis
- Python at JWST: Astronomical image manipulation (PyFITS)
- Python at INRIA: scikit-learn framework for machine/deep learning



Sustainability : Good packaging helps researchers to do more with less

Relevance for research and researchers

- **Reproducibility**
  - … to reproduce experimental results
  - … to ensure **consistency** in the **results** on **different** systems
- **Efficiency**
  - … to save time and reduce the risk of errors
- **Collaboration**
  - … to allow researchers to share their code with others
  - … to make it easier to collaborate together
- **Versioning**
  - … to allow researchers to follow the history of a project
  - … to compare multiple revisions of the same project

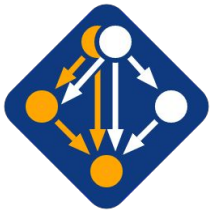
Motivation : desired properties

Software is getting more complicated

- **Lightweight and portable:** user should not have to spawn an **outdated** full-fledged container to replicate an environment which only requires small adjustments to his user paths
- **Productivity:** spending minimal time to replicate this environment
- **Reproducibility:** replicating an environment in different contexts
- **Observability:** understanding our ecosystem, e.g. measuring its health, size and other relevant characteristics

State-of-the-art

- Language-specific package managers: pip/npm/composer/…
- System package managers: apt/rpm/pacman/…
- Meta package managers : Conda/Nix/Spack/…
  - Conda (from Anaconda, a company): Python data science
  - Spack: HPC-oriented package manager
  - Nix: general-purpose meta package manager

In practice, artifact evaluation is done through VM/Docker/Singularity/OCI images

Research gap

- Academical evaluation of the prevalence of implicit dependencies over a collection of commonly used packages or specific package indexes is **limited**, e.g. PyPI: see Bezemer and al. (2017) and Ren and al. (2019) for recent results.
- No standard format to specify native dependencies metadata or cross-language dependencies, i.e. a language-specific package manager is good at one language at a time.
- Existing solutions to automatically figure out **environment** dependencies:
    - DockerizeMe (2019): strategies to render a Dockerfile based on existing metadata → static-only and incomplete
    - CARE (2014): System Call Interposition with *PRoot* → PRoot has heavy performance penalties
    - CDE (2011): System Call Interposition → ptrace cannot be called twice!

No general solution to (semi)-automatically figure out **build** dependencies!

Problem statement

How to develop a tool that dispenses automatically required build dependencies to a running program?

- … such that it can run on mainstream operating systems
- … such that it does not induce visible effects on the “run environment”
- … such that it does not require extensive heuristics for each “build environment”
- … such that it can be used to research and diagnose dependencies in ecosystems

System: An automatic native dependencies dispenser

A FUSE filesystem acting as a proxy to the database of Nix packages (nixpkgs)

System design goals:

- Portability
- Completeness
- Minimal performance overhead

Outline

- ~~Motivation~~
- Background
  - Build systems
- Design
- Implementation
- Evaluation

A primer on build systems

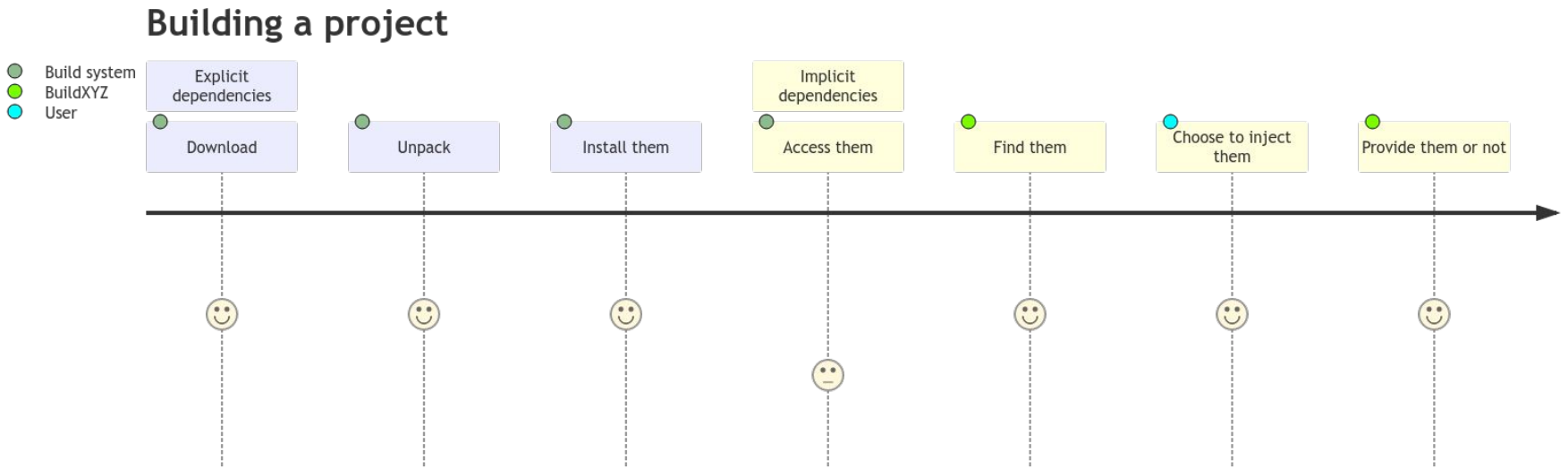
- Build systems are tools that automate the process of building software applications.
- Dependencies can be categorized: system or application dependencies.

- Build systems often use system environment variables to configure their behavior.
  - PATH environment variable : binaries
  - LD_LIBRARY_PATH environment variable : shared libraries
- Build systems sometimes use file systems accesses to test for available capabilities
  - … compiling a BSD specific program to test if we are on a BSD-based operating system

Outline

- ~~Motivation~~
- ~~Background~~
- Design
  - System design
- Implementation
- Evaluation

System overview

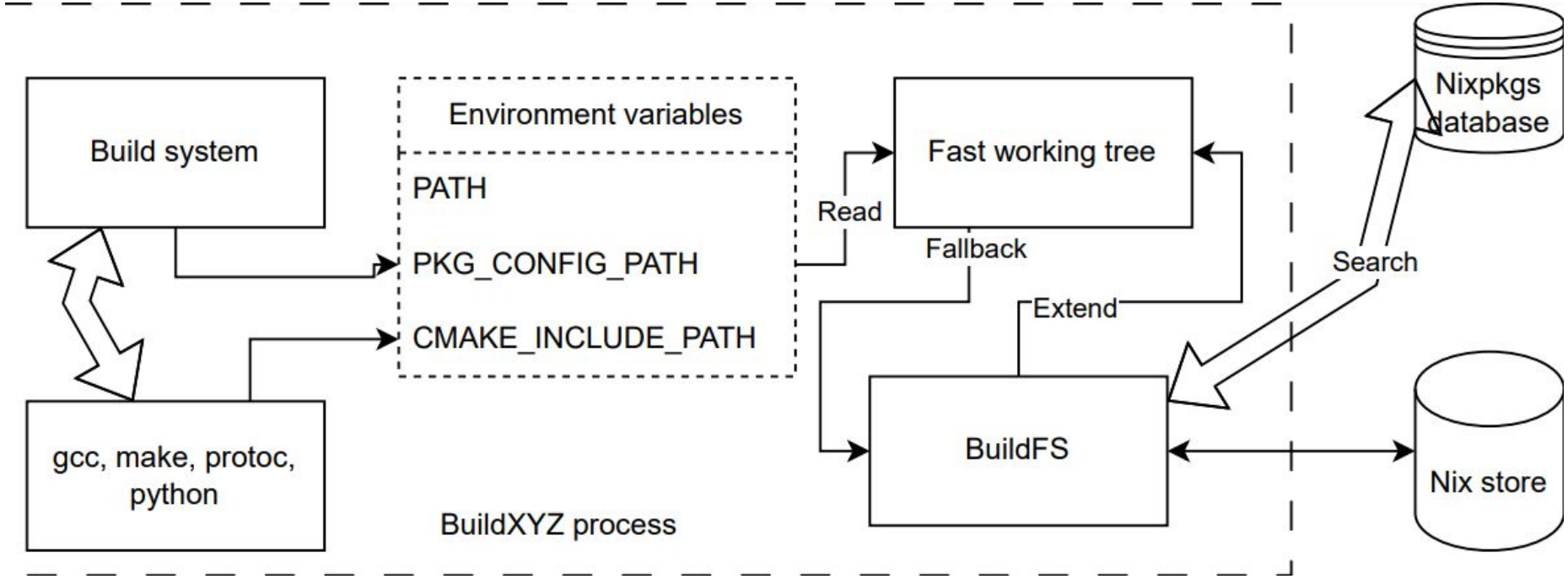


Design overview

- **BuildFS (FUSE) :** intercept **negative** lookups from the build system
  - … which queries the local software collection database about this path
  - … takes decision based on policy (user interaction, saved preferences, …)
  - … makes the path available on the local system (download, install, etc.)
  - … redirects the build system to a symlink to the path
- **Fast working tree (tmpfs + kernel filesystem) :** records seen dependencies
  - … consisting of a tree of symlinks previously returned to the build system during negative lookups
  - … to maintain low performance overhead for the same successive lookups

FUSE acts as the **last-resort** location for missing dependencies during build.

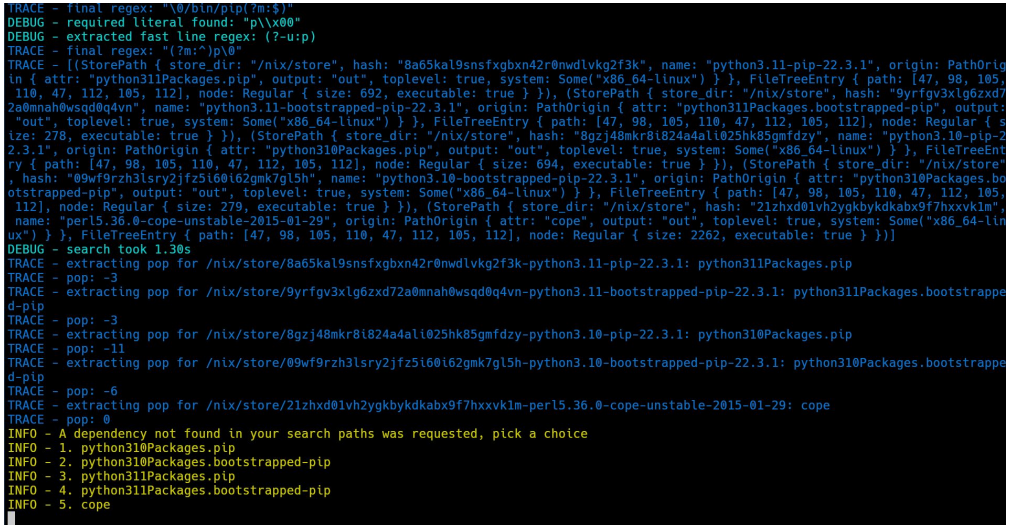
System overview



Incremental resolution of missing dependencies

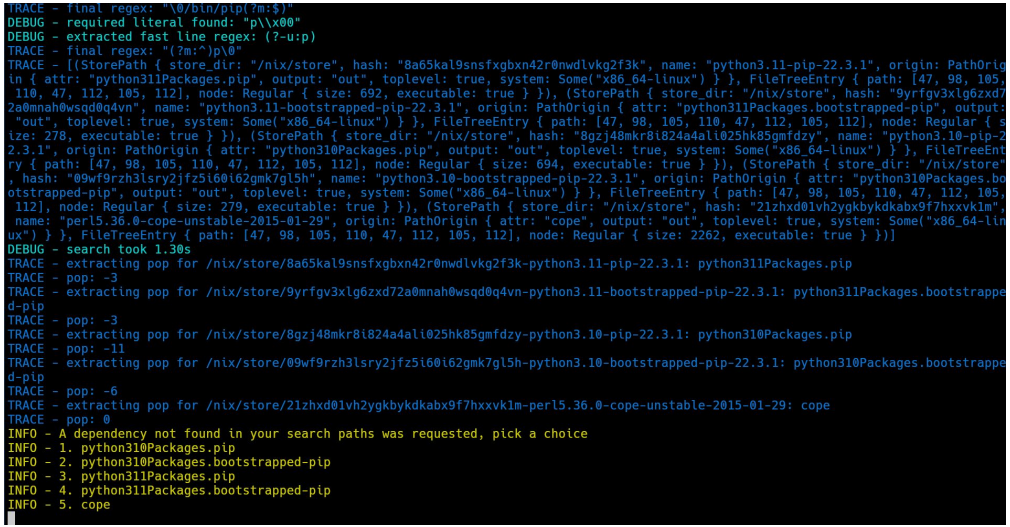
Overview by “example”

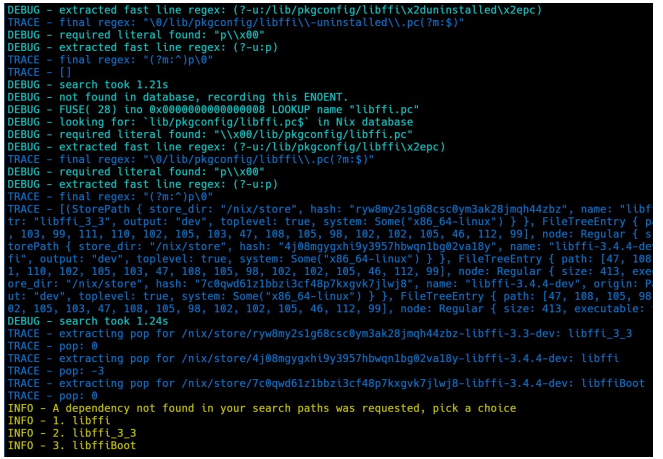
```
$ export PATH="/bin:/usr/bin:/fast/bin/:tmp/buildxyz.M2zHiSs7m8/bin"
$ export LD_LIBRARY_PATH="/usr/lib:/fast/lib:/tmp/buildxyz.M2zHiSs7m8/lib"
$ pip install cffi --no-binary :all:
```



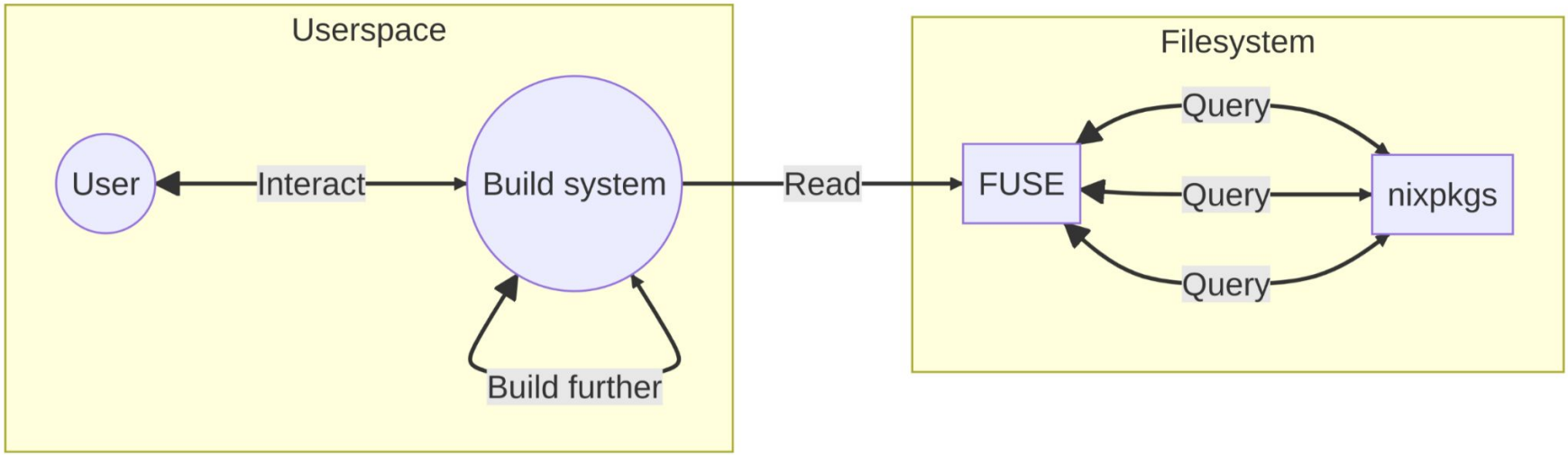
Overview by “example”

```
$ buildxyz “pip install cffi --no-binary :all:”
```





System overview



Incremental resolution of missing dependencies

Outline

- ~~Motivation~~
- ~~Background~~
- ~~Design~~
- Implementation
- Evaluation

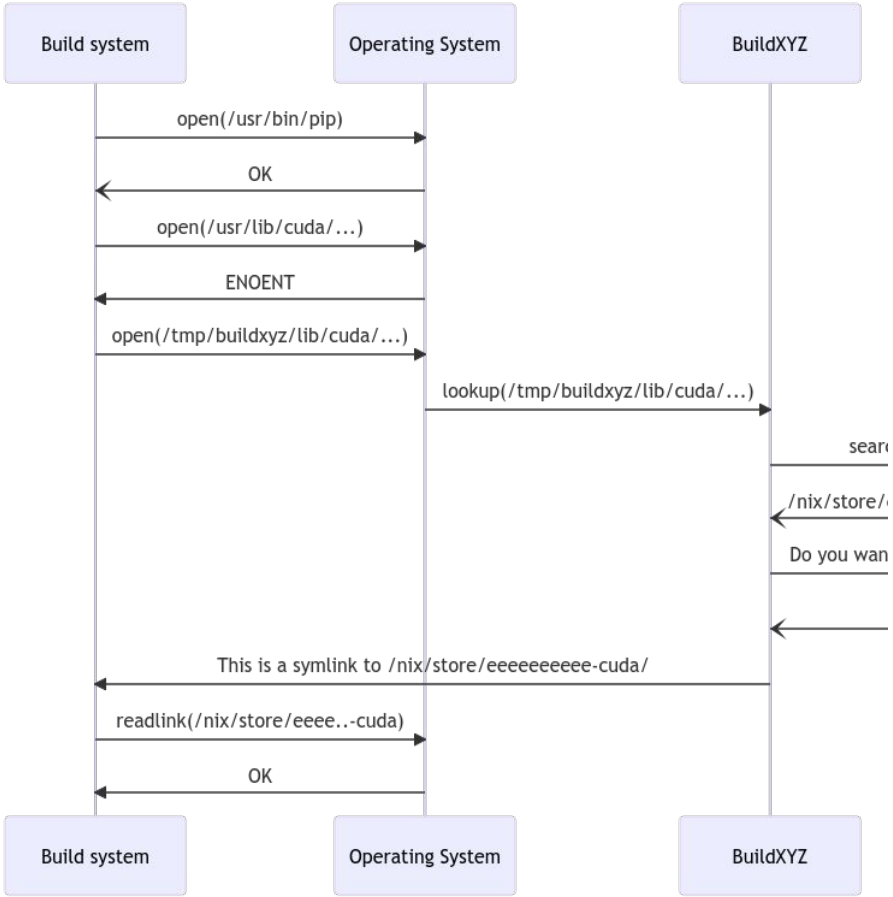
Implementation

BuildXYZ is built on FUSE and nixpkgs

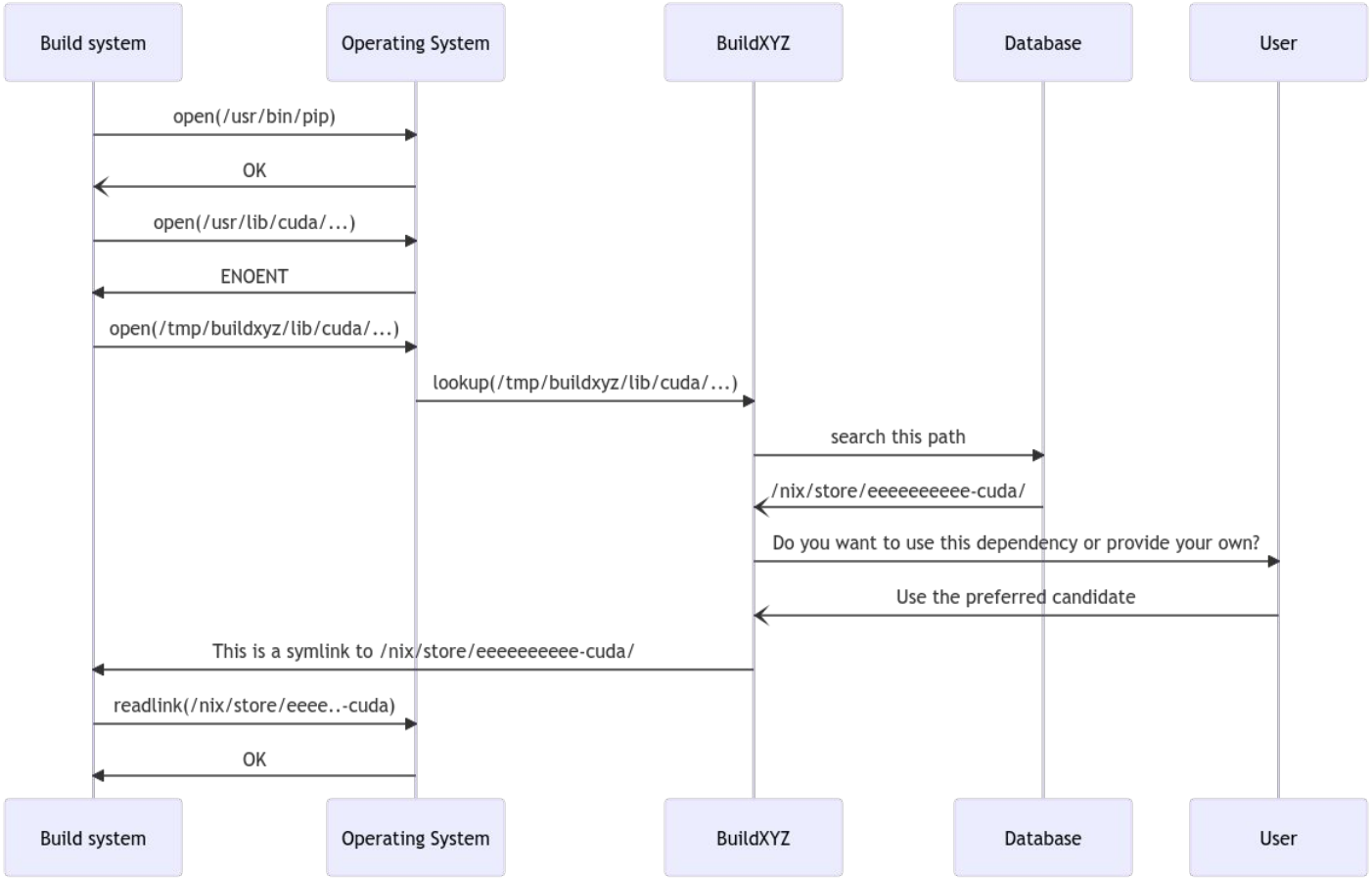
- **Completeness:** Uses a large database - nixpkgs[1] - 81.9K packages!
- **Performance:** Index all nixpkgs paths in a locate database - queries in 600ms!
- **Portability:** No system-call tracing - macOS support!

[1]Repology statistics: https://repology.org/

Implementation details : “which path are you looking?”



Implementation details : “which path are you looking?”



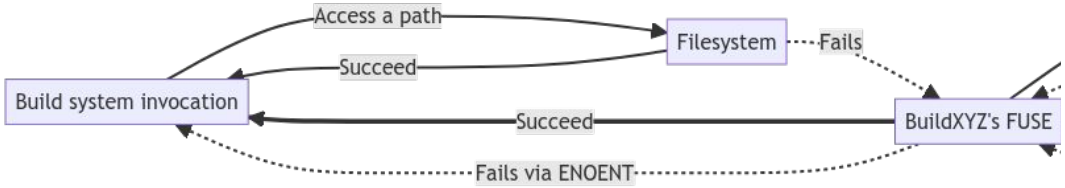
Implementation details : “record phase”

After a successful run, all “decisions” are recorded in a TOML file which can be **shared** and **analyzed** :

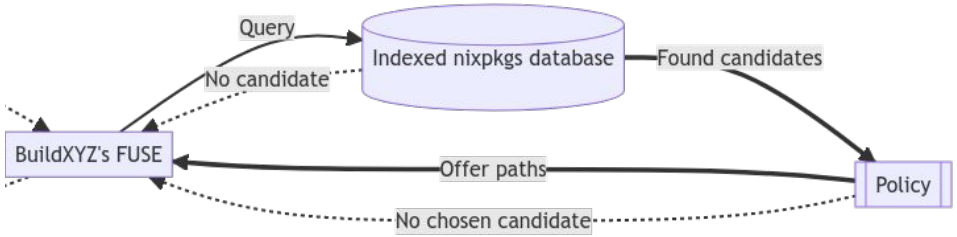
```toml
["lib/pkgconfig/libffi.pc"]
decision = "provide"
file_entry_name = "/lib/pkgconfig/libffi.pc"
kind = "symlink"
resolution = "constant"

["lib/pkgconfig/libffi.pc".store_path]
hash = "4j08mgygxhi9y3957hbwqn1bg02va18y"
name = "libffi-3.4.4-dev"
store_dir = "/nix/store"

["lib/pkgconfig/libffi.pc".store_path.origin]
attr = "libffi"
output = "dev"
system = "x86_64-linux"
toplevel = true
```

```toml
["bin/rustc"]
decision = "ignore"
resolution = "constant"
```

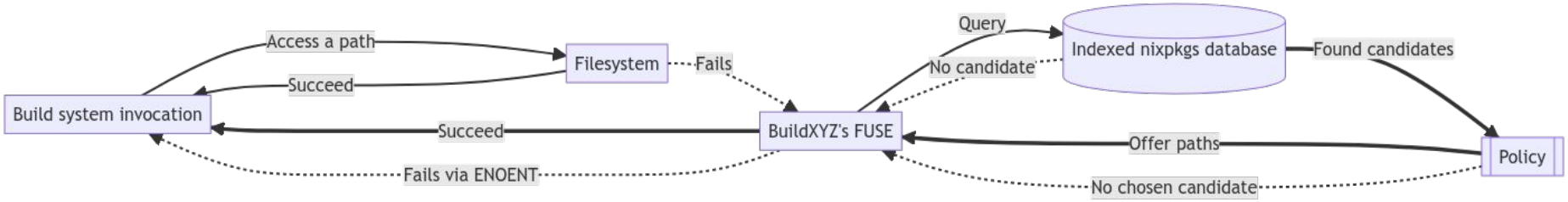
BuildXYZ dispensing workflow



BuildXYZ dispensing workflow



BuildXYZ dispensing workflow



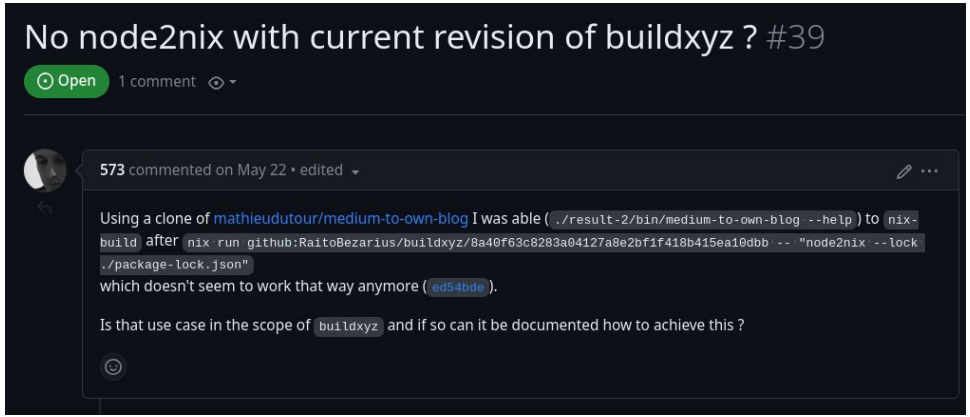
BuildXYZ's FUSE is a magic fallback location for missing dependencies wired up to nixpkgs

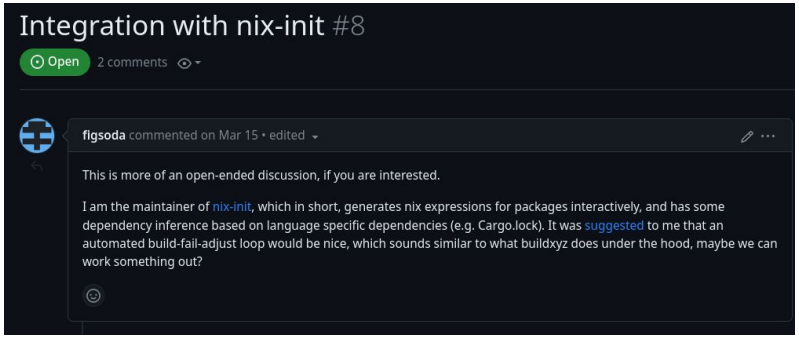
Outline

- ~~Motivation~~
- ~~Background~~
- ~~Design~~
- ~~Implementation~~
- Evaluation

Evaluation

- What is the coverage of buildxyz in fully automated cases?
  - … against PyPI top 5000 packages
  - … against commonly-used build systems: Meson, CMake, autotools among the Debian Popularity Contest
- Real world usage of buildxyz:





Remaining challenges before 1.0

- Symlink in the “fast extension” algorithm is complicated

- Test harness for native projects
  - detecting whether a project uses Meson, CMake or Autotools
  - buildxyz “the classical recipes for such a project”
  - sandboxing is important

Summary

Providing native dependencies for packages is still an unsolved problem

- no metadata for native dependencies
- empirically, packaging is a very messy suboptimal experience
- shipping the whole machine (i.e. Docker) is wasteful

BuildXYZ:

- portable via a FUSE approach
- high coverage through nixpkgs database
- renders implicit dependencies

Try it out!

https://github.com/RaitoBezarius/buildxyz