TUM

Technical University of Munich

RESEARCH INTERNSHIP REPORT

Advisors : Jörg THALHEIM, Pramod BHATOTIA

01/03/2023 - 31/07/2023, 5 months

Automatic on-demand dependencies dispenser

Ryan LAHFA

M1 DIENS

Contents

1 Introduction/Context 3
- 1.1 Taxonomy of packaging . . . 3
- 1.2 The (historically) missing standard for C/C++ . 3
- 1.3 Application-level packages and their relations with cross-language native dependencies . . . 4
- 1.4 C/C++ consequences on cross-language native dependencies in other languages . . . 4

2 Motivation 4
- 2.1 Problem statement . . . 4
- 2.2 Source-based or binary-based distributions . . . 4
- 2.3 Case study: scientific libraries . . . 5
  - 2.3.1 BLAS and LAPACK libraries . . . 5
  - 2.3.2 Geospatial stack: the GDAL case . . . 5
- 2.4 Source builds can be made as convenient as binary builds . . . 5

3 System overview 6
- 3.1 Problem statement and design goals . . . 6
- 3.2 BuildXYZ overview . . . 6
- 3.3 System components . . . 6
- 3.4 Design challenges & Key ideas . . . 7

4 Detailed design: Automatic dispensing of dependencies 8
- 4.1 System call interposition: ptrace . . . 8
- 4.2 Passive monitoring: SystemTap / eBPF . . . 8
- 4.3 Custom filesystems: FUSE . . . 8

5 Detailed design: custom filesystems 8
- 5.1 Abstract modelling of build system requests . . . 8
- 5.2 Relations between POSIX system calls, Linux virtual filesystem, and filesystem primitives . . . 9
  - 5.2.1 FUSE specifics . . . 10
  - 5.2.2 General optimization: the fast working tree 10
- 5.3 Toolchain and cross-compilation . . . 11
- 5.4 Learn from user input: resolution recording . . . 11
- 5.5 Learn from failures: parsing compilation errors and backtracking . . . 12
- 5.6 Protecting the user during operations . . . 12

6 Empirical evaluation 12
- 6.1 nixpkgs as a dataset . . . 12
- 6.2 Experimental setup . . . 13
- 6.3 Install evaluation . . . 13
  - 6.3.1 Remarks on results . . . 13
- 6.4 Case studies . . . 13
  - 6.4.1 Automatic packaging . . . 13
  - 6.4.2 Lazy package management . . . 14
  - 6.4.3 Diagnostics of filesystem patterns . . . 14
  - 6.4.4 Run binaries without knowing the runtime dependencies . . . 14
- 6.5 Real-world deployment experience . . . 15

7 Related Work 15
- 7.1 On-demand dependency dispensing . . . 15
- 7.2 Language-specific package managers . . . 15
  - 7.2.1 The case for C/C++ . . . 15
  - 7.2.2 The case for Python . . . 16
- 7.3 Version constraints solving . . . 16
- 7.4 Automatic repair based on build breakage data . 16
- 7.5 Misc . . . 16

8 Conclusion & Future works 16
- 8.1 Fixing the missing data via materialization . . . 17
- 8.2 A common format for cross-language and native dependencies . . . 17
- 8.3 Stable ABI to interop language-specific package manager and system package manager . . . 17

Abstract

In this research paper, we address the issue of expressing dependencies on native libraries and build tools in modern package managers and build systems. Despite the existence of well-defined dependency list and version lockfiles in tools such as cargo, yarn, and poetry, there is still a gap when it comes to native libraries and build tools. C/C++ libraries are primarily managed by Linux distributions, and while build systems have checks to detect them, they do not have mechanisms to automatically install or provide them. This can also be a problem when high-level programming languages such as Python require these libraries for their own packages. The current best practice is to provide a textual description or tutorial that a user must manually follow to build the program, which is often error-prone.

To solve this problem in a more automatic way, we propose BuildXYZ, an interactive build environment that can automatically detect missing build dependencies. BuildXYZ mounts itself as a FUSE filesystem and extends environment variables commonly used by build systems (PATH, C_INCLUDE_PATH, LIBRARY_PATH) to point to this files system. BuildXYZ records file access when compiler, linker and build tools are looking dependencies in this filesystems and proposes the user automatically to add missing dependencies. It uses a file list database of 80,000 packages from nixpkgs and heuristics such as the build process accessing a library, header file, executable or libtool/.pc file to provide the necessary dependencies. Using a simple popularity ranking algorithm based on a dependency graph from the nixpkgs repository, it is able to converge faster by providing the most likely package first.

We have tested BuildXYZ on multiple popular projects based on the PyPI and RubyGems "most downloaded" dependencies and achieved encouraging results in the evaluation. While BuildXYZ does not fully replace human effort in packaging, it is a useful tool as it automates a significant portion of the time-consuming packaging process and provides insights on the packaging ecosystem as a whole.

1 Introduction/Context

1.1 Taxonomy of packaging

Software packaging ecosystems typically consist of package managers that are used to manage software packages and their dependencies. These package managers are often categorized based on their scope and functionality. In this section, we provide a categorization of packaging ecosystems based on their scope and functionality.

Type	Scope	Examples
System-level	System-wide	apt, pacman, nix
Application-level	User or project wide	pip, npm, gem
Organisation-level	Organisation or project wide	Bazel, Nix, Buck2

Table 1: Simple package manager classification

System-level package managers are used to manage system-level packages and dependencies. They are typically pre-installed on the operating system and are used to manage packages required by the system, such as libraries and utilities.

Examples of system-level package managers include APT, YUM, and Homebrew. APT is the default package manager for Debian-based Linux distributions such as Ubuntu, while YUM is the default package manager for Red Hat-based Linux distributions such as CentOS and Fedora. Homebrew is a package manager for macOS.

Application-level package managers, on the other hand, are used to manage packages required by specific applications. These package managers are typically language-specific and are used to manage packages required by the application to function properly.

Examples of application-level package managers include pip, npm, and Composer: pip is the package manager for Python. npm is the default package manager for Node.js. Composer is the package manager for PHP.

Finally, meta build systems are used to manage the entire build process, including the building of dependencies. These build systems are typically used in large-scale projects where there are many dependencies that need to be managed. Furthermore, they can understand and manage multiple languages at the same time and cross language dependencies.

Examples of meta build systems include Nix, Bazel, and Gradle. Nix is a meta build system that is used to build and manage software packages and their dependencies. It allows for the creation of reproducible builds, where the exact same build can be replicated across different systems. Bazel is a build system that is used to build and test software. It is designed to handle large-scale projects with many dependencies and supports a wide range of programming languages. Gradle is a build system that is used to build and manage software projects. It is primarily used for building Java applications, but it also supports other programming languages such as C++ and Python.

In summary, packaging ecosystems and their respective package managers are an important part of software development. System-level package managers are used to manage system-level packages and dependencies, application-level package managers are used to manage application-specific packages and dependencies, and meta build systems are used to manage the entire build process, including dependencies. Table 1.1 provides a summary of the categorization of packaging ecosystems based on their scope and type.

1.2 The (historically) missing standard for C/C++

In this section and the next ones, we will develop an history of why the coupling between system-level and application-level became complicated.

C and C++ do not have a built-in package manager because they were designed to be low-level languages that provide direct access to system resources. This means that they do not have a standard library or runtime environment that can be easily managed by a package manager.

However, there have been attempts to introduce a module system in C++ to make it easier to manage dependencies and reuse code. The latest proposal for modules in C++ is the C++20 module system, which allows for the creation of modular code that can be compiled separately and linked together at runtime. This proposal aims to improve the performance and maintainability of C++ code by providing a standardized way to manage dependencies and reduce the need for header files.

One of the current attempts to have packaging information for C++ libraries is the use of pkg-config. pkg-config is a tool that provides a standardized way to retrieve information about installed libraries, including their version, compiler and linker flags, and other metadata. It is widely used in the Linux and Unix ecosystems to manage dependencies and build software.

To use pkg-config with a C++ library, the library must provide a .pc file that contains the necessary metadata. This file typically includes information about the library's name, version, dependencies, and installation paths. When a user wants to build software that depends on the library, they can use pkg-config to retrieve this information and pass it to their compiler and linker.

Another attempt to have packaging information for C++ libraries is the use of CMake, a cross-platform build system that can generate build files for a variety of compilers and platforms. CMake provides a way to define dependencies and build targets for C++ libraries, and can generate configuration files that can be used by other build systems or package managers.

Finally, C/C++ package managers like Conan have seen the day and are popular in large scale C++ projects, some meta build systems even integrate hooks to support the Conan package manager out of the box.

Overall, while there is no standard packaging system for C++

libraries, tools like pkg-config, CMake, Conan provide a way to manage dependencies and build software in a standardized way.

1.3 Application-level packages and their relations with cross-language native dependencies

Application-level packages, for many reasons, e.g. acceleration, interfacing with system APIs, will make use of special language-specific dependencies which are thin wrappers on the top of a system library coming from another language.

We call such dependencies “cross-language native dependencies”, as an example: a Python’s native C extension is a cross-language native dependency written in the C programming language providing bindings to access C code from Python. Another example could be a Python’s native Rust extension written in the Rust programming language.

Many interpreted languages make use of cross-language native dependencies for agility, performance and taping into the safety’s properties of the cross-language, e.g. Rust’s borrow checker here for memory safety.

1.4 C/C++ consequences on cross-language native dependencies in other languages

As C/C++ is the de-facto the classical system programming language for cross-language native dependencies, a lot of dependencies depend on C/C++ libraries, even if the cross-language dependency is written originally in Rust, it might itself depend on OpenSSL which is written in C.

Therefore, C/C++ is currently one of the most important language to write cross-language native dependencies and has impact on all other ecosystems, including other native ecosystems.

Given the fact that C/C++ does not have any package manager, this problem bubbles up in all ecosystems, including Rust, which can be seen with some dependencies vendoring a copy of the C library and recompiling it internally and offering sometimes only ways to use `pkg-config` to reuse an existing local library.

2 Motivation

2.1 Problem statement

Figuring out the dependencies of any project is usually communicated via: "dependency files" (`requirements.txt`), "lock files" (`poetry.lock`, `package-lock.json`) and finally "human instructions" (`README.md`).

It is difficult to achieve perfect instructions that works across all reasonably supported systems, because authors of a project are not packaging expert matter and does not always exactly the full dependencies of their project in certain dependencies or does not know the full capabilities of other platforms that they are not used to develop against.

Therefore, most projects provides partial information on the actual dependencies of a project and often misses the implicit dependencies: system-level dependencies, cross-language dependencies, to give an example.

In addition, we will try to make it clear when we talk about build dependencies, i.e. dependencies required to build the project but not to **run** the program it, vs. runtime dependencies, i.e. dependencies requires to **run** the program. Build dependencies are therefore **runtime dependencies** of the build process of a given program.

Here, we aim to automatically derive most, if not all, dependencies of any given project based on their runtime patterns, including **implicit** ones.

We will see through multiple case studies why this is of major importance for research and industry.

2.2 Source-based or binary-based distributions

In Python Enhancement Proposal (PEP) 0427 about "Wheels", accepted in 2013, the following motivations are written:

> Python needs a package format that is easier to install than sdist. Python’s sdist packages are defined by and require the distutils and setuptools build systems, running arbitrary code to build-and-install, and re-compile, code just so it can be installed into a new virtualenv. This system of conflating build-install is slow, hard to maintain, and hinders innovation in both build systems and installers.

Wheel is a self-contained ZIP format representing a Python dependency including its own binaries dependencies, the first problem is that wheels are ABI specific, so a specific wheel has to be built for each ABI.

This is the rationale for the creation of PEP513, PEP571, PEP599 and PEP600, a proposal to enable to ship glibc versioned wheels. As glibc is not forward compatible, it is crucial as an author to build wheels on the oldest glibc version you can reasonably use.

Unfortunately, as the PEP600 mentions it:

> Non-goals include: handling non-glibc-based platforms; integrating with external package managers or handling external dependencies such as CUDA; making manylinux tags more sophisticated than their Windows/macOS equivalents; doing anything besides taking our existing tried-and-tested approach and streamlining it. These are important issues and other PEPs may address them in the future, but for this PEP they’re out of scope.

A lot of things remains out of scope and the value of wheels, as they are author-created, depends directly on multiple factors:

- **Trust in author**: author could push a wheel that does not match the source code or author’s credentials could be stolen to push unauthorized updates

- **Trust in the package index**: package index is not transparent in build reproducibility so alteration in what the author has pushed cannot be detected
- **glibc dependency**: musl is supported to some extent, the rest must be handled by user directly
- **native dependencies**: native dependencies are completely ignored to some extent
- **toolchain and cross compilation**: this aspect is completely ignored, gcc/llvm differences are not considered and cross-compilation contexts are out of scope

It appears that the value of wheels come from the difficulty for a package manager to detect, suggest adequately for native dependencies, taking into account glibc alternatives.

Overall, wheels present severe shortcomings whenever you are in one of these non-trivial contexts: cross-compilation, non-standard glibc, native system dependencies, package variants.

Package authors sidestepped many of these issues by cleverly exploiting Python semantics, i.e. publishing a package variant under another name and their wheels.

Nevertheless, because of those severe shortcomings and the inevitability that wheels have a blind trust model in the authors and the package index, we believe that source-based builds combined with an adequate caching model remains the way to go, e.g. what Nix offers with the Nix store layer [10].

Tangentially, Conda, the open source, cross-platform, language-agnostic package manager and environment management system works in a similar way, with a corporate backing, they are also very popular in the data science ecosystem in Python. Contrary to pip's Python package manager, they packages "end to end" the required dependencies, including ones.

A simple query via Sourcegraph: https://sourcegraph.com/search?q=context:global+conda+install+file:README.md+count:all&patternType=standard&sm=1&groupBy=repo shows that around 38 600 repositories suggests `conda install` in their instructions, i.e. `README.md`.

In the next section, we will examine two major examples in the scientific ecosystem.

2.3 Case study: scientific libraries

2.3.1 BLAS and LAPACK libraries

BLAS and LAPACK are a set of scientific procedures providing high performance for scientific computations, they are prevalent in many downstream packages, e.g. NumPy, SciPy, etc.

BLAS and LAPACK only defines a "generic interface" and they can be swapped out for different implementations depending on the target platform: some vendors provide CPU-specific accelerated BLAS such as Intel MKL, AMD BLIS for example. There is also a reference BLAS called netlib and vendor-neutral accelerated BLAS called OpenBLAS.

We mentioned in 2.2 that Python packages are usually distributed via wheels and therefore they vendor their own copy of BLAS, often using OpenBLAS. In turn, this renders almost impossible the capability of replacing a BLAS with another because you need to go back to a source build to use it.

Therefore, due to the choice of vendoring those libraries, it is complicated to tap into the full agility enabled by the BLAS/LAPACK variant models as a user rarely knows how to recompile a given package with a choice of BLAS given that application-specific package manager will recommend wheels, furthermore, given there is no metadata about this type of variants, it is difficult to express a preference on a BLAS, leading to complicated installation procedures whenever someone wants to make use of them.

A way to side-step this is to **never use** any Python wheels and prepare your environment so that the build scripts used by those projects pick up your BLAS but this is highly error-prone and has to be done for each project using those packages.

2.3.2 Geospatial stack: the GDAL case

Another set of important libraries in science is the geospatial stack, among which, we can find the GDAL library, a translator for raster and vector geospatial data formats.

GDAL is a C/C++ project with multiple optional native dependencies, again, the wheel model here does not make a lot of sense as it would require to rebuild multiple times GDAL for all variants and vendors them inside of a wheel, though the package management cannot express the preference.

On the PyPI package README page of GDAL: https://pypi.org/project/GDAL/, this can be read, at the time of writing:

> GDAL can be quite complex to build and install, particularly on Windows and MacOS. Pre built binaries are provided for the conda system: [...]

2.4 Source builds can be made as convenient as binary builds

Given that application-level package managers did not take into account the potential richness of their own ecosystems, it seems difficult for them to retrofit all of these features, especially given that some large-scale ecosystems changes already happened in Python for example with `pyproject.toml`.

In this work, we advocate to abandon binary builds as they are provided by an application-level package manager : they are too brittle to tap into the capabilities of the software ecosystem.

Instead of that, we advocate to ensure that source-first builds always work and invite ecosystems to leverage existing cache semantics to provide for binaries builds.

The question regarding whether the binary in the cache should be provided by the author, or should the cache be provided by the language ecosystem infrastructure, etc. are out of scope here.

NixOS, for instance, offers a default `cache.nixos.org` relying on the artifacts of the build farm `hydra.nixos.org` for many projects, including `nixpkgs`.

Users can run their own Hydra instance and their own cache instances and configure their system to use it, ignore the default one, etc.

Therefore, even if nixpkgs is a source-first distribution, it can, in almost all cases, provide binaries to end-users and offer escape hatches for any user wanting more by falling back to a source build.

Furthermore, this is not a NixOS specific feature as Nix can run on many Linux distributions along the system-level package manager and could be made run on other non-Linux operating systems.

Finally, we will use those features of Nix to offer seamless dependencies, sometimes already cached, sometimes not which will be compiled transparently for the final user.

3 System overview

3.1 Problem statement and design goals

While designing BuildXYZ, we set the following goals:

- **Cross-platform:** Our tool should not depend on platform-specific semantics to ensure it can be used on many development platforms, e.g. macOS, Windows, Linux.
- **Completeness:** As long as we can discover dependencies via file system accesses, we should be able to theoretically learn about all potential dependencies required by any build system supported by our hypotheses. Therefore, no extra instruction should be needed at all.
- **Efficient:** Our tool should not incur significant performance penalty to filesystem accesses. For this, our target is to ensure we do not incur much more extra time overall at the end of the build system process, which might perform slow operations like ad-hoc compilation or network operations.

3.2 BuildXYZ overview

The underlying intuition behind BuildXYZ is that most tools including build systems and compilers finds their dependencies via **search paths** which are discovered via **environment variables**. Then, **search paths** are hit one by one through filesystem accesses. Thankfully, most of the tools do not **walk** a search path which would generate a lot of spurious filesystem access, rather, they will directly access the required dependency by constructing a predefined path using search paths. Therefore, it is possible to provide dependency on-demand efficiently by adding special locations in those **environment variables**.

We present BuildXYZ an automatic dependency dispenser relying on FUSE to answer to requested paths which might be missing on the disk, which extends a "fast working tree" to avoid paying the FUSE penalties whenever a new *transitive closure* of a dependency has been found for the project.

BuildXYZ workflow. BuildXYZ requires very few changes to the existing workflows. Any build system invocation can be wrapped in a `buildxyz "existing build system invocation"`, e.g. `buildxyz make`. BuildXYZ will execute the following steps:

1. *Read existing resolutions databases.* Resolutions are pieces of knowledge which can be local to a language ecosystem, a project which explains how to automate a choice. They are useful whenever some choices are ambiguous, e.g. which version to use of the Boost C++ library. Resolutions have a priority system and can be overridden following a search path mechanism.
2. *Launch the BuildFS.* A BuildFS is initialized by spawning a FUSE process and creating Filesystem Hierarchy Standard directories as empty directories, it also initializes an indexed Nixpkgs database to search quickly for new paths.
3. *Construct the fast working tree using default resolutions.* A fast working tree is a on-disk location consisting of all already discovered dependencies which should not be served by BuildFS.
4. *Spawn the instrumented program.* Preparing the new environment by appending to each known environment variables, see table 3.2, our two special locations: one for the fast working tree and one for the BuildFS, in this order, to ensure fast working tree has priority.
5. *Whenever an unknown dependency is encountered, i.e. a filesystem access in the BuildFS.* We will look up our policy for this path and apply the decision: provide it or ignore it (answering `ENOENT` via the filesystem), here, we will reuse our resolution database for automation, relying on user input as a last resort. At the end, we will **record** this decision in **memory**.
6. *At the end of the instrumented program, new resolutions are recorded.* We write all new in-memory resolutions on disk for further reuse, e.g. distribution on version source control, processing with other tools, reuse with BuildXYZ, if the user asks for it.

3.3 System components

At a high-level, BuildXYZ's design relies on four components, as shown in figure 1.

- **Component #1: BuildFS** BuildFS is the BuildXYZ's FUSE filesystem which will handle all negative lookups of the filesystem which were not handled by the fast working tree, a fast on-disk filesystem "cache". As it is supposed only to be used when a new dependency is met, it will trigger queries to the Nixpkgs database and realize the selected dependency in the Nix store.

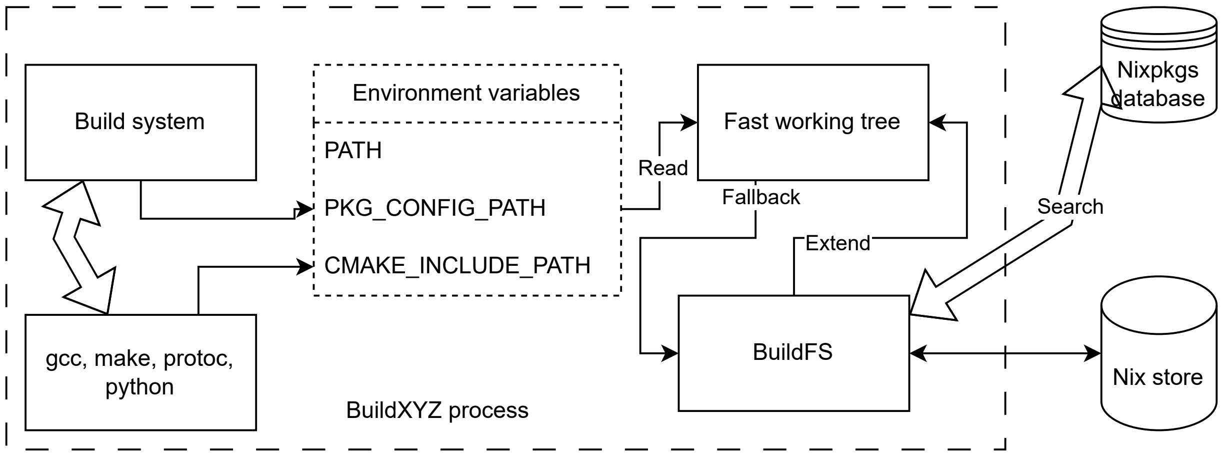

Figure 1: BuildXYZ high-level overview

Environment variable	Purpose	Always appended?
PATH	Location for binaries	yes
PERL5LIB	Perl libraries	no
PKG_CONFIG_PATH	pkgconfig .pc files	yes
CMAKE_INCLUDE_PATH	find_file, find_path CMake specific locations	yes
ACLOCAL_PATH	Automake macros locations	no
LIBRARY_PATH	Build-time libraries	yes
NIX_CFLAGS_COMPILE	Build-time include libraries	no

Table 2: Instrumented environment variables

- **Component #2: Fast working tree** The fast working tree is supposed to be a temporary filesystem location which contains the transitive closure of all dependencies met, symlinked to the Nix store. It is responsible to handle the "fast path" of dependencies dispensing.
- **Component #3: Nixpkgs** Nixpkgs is the database of "all potential dependencies" we can dispense for, as it is the largest collection of software in the world. We use Nix to do the heavy lifting regarding downloading, preparing and installing any path we need, this operation is called "realizing" a Nix store path. We pre-index a revision of Nixpkgs using nix-index to search efficiently for any binary, library or file we might need during build system operations.
- **Component #4: BuildXYZ process** BuildXYZ process is responsible for managing the lifecycle of the previous components: it will prepare the environment by injecting the instrumentation environment variables as shown in the figure, will start BuildFS, will load a fast working tree by reading the current available resolutions, run the build system and handle user interaction requests.

3.4 Design challenges & Key ideas

Automatically dispensing dependencies presents multiple challenges regarding the reality of packaging and build systems:

1. **False positives:** Sometimes, a build system will query the existence of the file for the sake of detecting another variable, e.g. "which operating system am I running on?", this is the case for example with `gstrip` which is the GNU version of `strip` and is classically available on BSD derivatives as two different binaries. Linux distributions usually do not provide it because it could confuse some build systems, which would believe running on BSD. Sometimes, **not providing** a dependency, even if we have it, is the right solution. A more general instance of this problem is autotools-based system which will try to perform certain actions for autodetection of supported features on the current system.
2. **LD_LIBRARY_PATH:** Providing for missing dynamic libraries is not easy because the way the dynamic loader works is by looking in `DT_RPATH` first then `LD_LIBRARY_PATH` then `DT_RUNPATH`, meaning it is not possible to keep the already-provided libraries via `DT_RUNPATH` (`DT_RPATH` has been deprecated) while providing for missing ones.
3. **Filesystem paths:** When implementing a FUSE filesystem, we really know parts of the filesystem path that is being accessed in the `lookup` API call. But we really want to know things like what is being accessed is `bin/curl` and not `bin` followed by `curl`. We work around this problem by rebuilding the whole filesystem hierarchy standard [41] "virtually" as fake inodes and track where are we at each lookup call. At the end, we will present every file as a symlink to the Nix store which will triggers a readlink to the Nix store's filesystem implementation.
4. **Nix:** When selecting a dependency, what is desired is to

pull the whole transitive closure to ensure its compatibility, most package managers relies on the assumption that the operating system's package manager will ensure a sort of coherence at the system-level using Filesystem Hierarchy Standard. In our experience, this does not happen as the software ecosystem is getting more complicated, therefore we lean on Nix's ability to provide us with "transitive closure" of software where we have the theoretical guarantee that any path has been compiled with its parents, guaranteeing its compatibility.

4 Detailed design: Automatic dispensing of dependencies

One of the challenges in packaging software is managing dependencies, particularly in languages like Python and Node.js, which have a vast number of third-party packages with complex dependencies. Traditionally, packaging systems install all dependencies at once during installation, leading to large installation times and potentially unnecessary disk space usage.

Furthermore, "all dependencies" is often defined via human-written instructions such as README or application-specific dependency manifest, e.g. `requirements.txt`. Our approach is to use an on-demand dependency dispenser to ensure we provide the **required** dependencies at every moment.

This approach has multiple advantages, inside of a given project :

- Minimization of the transitive closure of dependencies size
- Sidestep the system-level dependencies potential conflicts by performing local scoping of dependencies

can be used to further minimize the transitive closure of dependencies size of a project.

4.1 System call interposition: ptrace

System call interposition can listen to any read system call and provide on-demand the path, the idea of doing filesystem development using ptrace is described in [42].

The issue is that system call interposition is not available on all platforms, it requires some privileges and will prevent further system call interposition on the target, e.g. `strace` will be non-functional.

Therefore, we will not pursue this avenue, though [19] and [23] tried to provide a on-demand dispenser via ptrace which also repackaged all the read traces inside the final binary, nevertheless, section 4.2 of [17] explains some problems with listening to filesystem accesses via system call interposition.

4.2 Passive monitoring: SystemTap / eBPF

Another approach is to not interact directly with the software trying to install but passively listen to what happens and note the failures related to missing dependencies, as long as the software doesn't execute successfully, re-execute it.

The issue with this approach is that passive monitoring facilities like SystemTap [22], [13] or eBPF are not available on all systems, it requires also privileges to load such programs or scripts and will require many restarts of the program which can cause redownloads of assets.

As a result, we will not pursue this avenue neither, though we were not able to find in the literature any prior art on trying this method to learn about implicit dependencies, we believe it would probably be difficult as re-executing a binary can be expensive, though we believe that extending filesystems in userspace could also be achieved via [5] with a eBPF component.

4.3 Custom filesystems: FUSE

The approach we will take here is to provide a custom filesystem we have control over, via FUSE, which is known to work on Windows (WinFsp), macOS (macFUSE), Linux (FUSE), NetBSD (PUFFS) at least.

Furthermore, this does not require any kernel modification, the filesystem runs in userspace and can be further accelerated with passthrough techniques.

We summarize here the different approaches in automatic dispensing of dependencies.

5 Detailed design: custom filesystems

5.1 Abstract modelling of build system requests

We represent a build system by a $P = \{p_1, \ldots, p_n\}$ where each p_i is a different process, we can assume that each thread of each process is a process so each process has only thread really for simplification.

Each p_i inherits from the E_{BuildXYZ} environment variables which are instrumented for the build.

More precisely, for each search path (binaries, libraries, etc.), we proceed to append our BuildXYZ mountpoint :: Search path specific prefix to the end of this search path.

For example, for `PATH`, we append `BuildXYZ mountpoint :: bin` where `::` is the system path separator.

A build system can do many things: network access, filesystem access, general computations. Here, we are only interested into filesystem accesses that uses the ambient search paths.

When a p_i tries to find a resource q in one of the search path $S = \{s_1, \ldots, s_m\}$ where s_m is the BuildXYZ instrumented path.

Either it find q in s_i for $i \in [[1, m-1]]$, either it tries to search for q in s_m.

Remark : Notice here that if we reduce S to $\{s_m\}$, we will never rely on the user-provided environment, therefore, ensuring a higher level of purity of the results, at the cost of potentially having to review for more choices.

Searching for q in s_m means that we search q in our database of known paths, we rank the list of results according to a ranking policy $R : \text{Packages} \to \mathbb{R}$.

At that moment, multiple approaches are possible according to a selection policy $Q : \mathcal{P}(\text{Packages}) \to \text{Packages}$, we describe some of them which can be composed:

- either, we reuse an existing choice (transitive closure of already made choices)
- either, we enable the user to choose explicitly for its package (interactive) and record it for this session or on the disk

In this abstract modelling, we exclude build systems mechanism that relies on walking directories to discover available dependencies.

5.2 Relations between POSIX system calls, Linux virtual filesystem, and filesystem primitives

Previously, we described a set of abstract filesystem primitives, we will now map them over the Linux kernel primitives, without any loss of generality for other platforms as FUSE semantics relies on those primitives anyway.

Let be q a path requested by the build system and $S = \{s_1, \ldots, s_m\}$ a search path in which q will be looked in. Each item the build system will look for q, the build system will perform a filesystem access to `s_i/q` until there is a $i \in [[1, m]]$ that will provide this file, e.g. the filesystem does not return `ENOENT` for that path, this is what we call the "search path semantics", a build system that performs this is called a "search path abiding build system".

Note that some build systems may attempt to record which s_i worked and reuse it across execution which can be problematic because it may be the case that some s_i provides a path and a s_j for $j < i$ contains a path that no s_k for $k \geq i$ provides, we consider that such build systems does not honor rigorously search path semantics but our system will still work with a performance hit with them.

Let's assume that each try will be represented by a `open` system call on the path. As Linux implements a directory entry cache (dcache), the virtual filesystem layer will split the path to find the leaf directory entry. On the way, it will create any required directory entry and load the relevant inode. Whenever a new inode needs to be looked up by the virtual filesystem layer, the filesystem implementation `lookup` function will be called with the parent directory inode and the next path part.

Therefore, the filesystem implementation never sees directly the complete path, but only path parts and parent inode at a given time.

As said previously, requesting a path by the build system is not necessarily directly represented by a `open` system call on the path, sometimes, it is possible that a walking procedure has to be performed to load all available files before opening the relevant ones. Those situations are particularly problematic as there is no canonical representation of a subdirectory like `/usr/lib` because there are multiple variants and versions of the same libraries that can be offered as long as the build system `open` a precise path.

Nevertheless, assuming we get `lookup` function calls, our approach is to slowly build in-memory a view of an abstract filesystem:

- Pre-allocate filesystem hierarchy standard [41] well-known inodes, e.g. `/bin`, `/include`, `/perl`, `/aclocal`, `/cmake`, `/lib`, `/lib/pkgconfig`
- Maintain a in-memory hash map of inode to filesystem paths
- For each `lookup` call, find the parent inode's filesystem path which we call the "parent prefix" and concatenate it with the current path segment to form the "real path"

The previous procedure make it possible to recompute q in the filesystem implementation code, an alternative is to notice while servicing the filesystem request, the caller process is in a uninterruptible sleep state waiting for a I/O system call, this is always possible because `pid_nr_ns(task_pid(current), <a pid namespace>)` will return the task's PID waiting for the system call in the specific PID namespace being used. Once, the PID is obtained, the procfs expose `/proc/$pid/syscalls` which contain the currently running system call with its arguments, it can be parsed and used to determine what was the original argument passed to the `open` or similar system call to find about the complete path directly.

In our case, we do not adopt this strategy because even if we knew the full path and determined if q was present or missing, we would not be able to reply an `ENOENT` error before reaching the leaf node.

By now, we know q in the filesystem implementation code, so we can search for q:

- Either, q is a known global directory, we serve it because it is already pre-allocated
- Either, q is a unknown global directory, we reply `ENOENT` as we want to ascertain a minimal level of purity
- Either, q is a previously encountered non-existent path, we reply `ENOENT` directly rather than searching for it again
- Either, q is part of an previously encountered dependency, we verify if `on-disk cache :: q` exists and we redirect to the on-disk cache directly
- Otherwise, we will search for q in the indexed database of paths

Searching in the indexed database of paths requires to determine a proper search query, we use regular expressions and look for `^/q$`, it is particularly useful to narrow down to candidates with a high chance of being the dependency the build system is interested in, e.g. `^/lib/libboost_math_c99\.so\.1\.75\.0$` ensures that we will not have unexpected directory structure like `/usr/lib/...`, indeed, as more and more software decides to vendor their own dependencies, we could be lured into grabbing the dependency of a full application which could work but does not represent the "library" dependency, achieving poor disk space usage or compilation times.

Finally, we use the Q function introduced earlier on the set of results to choose the candidate which we will provide in our filesystem, we summarize the whole dispatch process in the figure 2.

5.2.1 FUSE specifics

We decided to use a Filesystem in Userspace (FUSE) implementation rather in-kernel implementation, there are multiple reasons for that:

- FUSE makes development faster and easier at the cost of performance penalties
- FUSE makes the filesystem safer to trust for end-users as its actual implementation lives in userspace and is mediated by the FUSE kernel APIs
- `BuildFS` can be combined with remote servers to offer a shared negative lookup filesystem inside the same organization unit

Furthermore, we overcome FUSE performance penalties by noticing that as long as we stay faster than the build system and package management operations like download (network), the end-result will not be noticeable to the end-user.

Moreover, FUSE is actively developed and extended, in the future, it may be possible to explore the usage of FUSE passthrough to make FUSE being used only for `open` system calls and let the kernel reuse an on-disk filesystem for the rest. With the ebpf-fuse, it may be possible to include the `open` logic inside an eBPF program which will forward directly the I/O requests to lower filesystem in the kernel.

5.2.2 General optimization: the fast working tree

Going back to the initial description, if we look at what q could be in the context of a build system, e.g. a dynamic library, a C header, a binary, if we can assume further that q is not an isolated path and has siblings paths, e.g. another relevant dynamic library or C headers or binaries, therefore, it makes up what we usually call a "package" formed of multiple paths.

If we assume that the build system will not ask only for q but some siblings of q in the future, it may be of interest to leverage this possibility directly when a package candidate is chosen in the filesystem code, i.e. whenever we extend our working tree of filesystem nodes by q, we will actually extend it by all the components of a package containing q chosen during candidate selection phase.

Note that there is no "the package composed of q", because there are many packages formed of q, but there is only a unique package which is chosen during execution of the program, we may refer to "the package" as this chosen package during execution, but there is a policy choice dependency in what we call "the package".

Also, instead of creating the inodes in our custom filesystem, we can pre-allocate at program startup a special directory "tree" on a location of user's choice, ideally a temporary fast location, e.g. `tmpfs`, where we will recreate the filesystem hierarchy standard structure with symlinks, let us denote that special directory tree: the fast working tree, that is, a working tree that is faster than the custom filesystem tree.

So back to $S = \{s_1, \ldots, s_m\}$ our search path for a given build system, we will ensure that S contains the fast working tree location **then** the custom filesystem location.

This way, when q is looked via the custom filesystem, we extend in the fast working tree location, on the next lookup of a sibling of q by the build system, it should end up in the fast working tree instead of asking again the custom filesystem implementation for that file.

This optimization is not only a computation optimization but achieves increased correctness, indeed:

Whenever two paths are encountered with a search path abiding build system, if one of the path is a sibling of the other, we want to ensure we select always the same package otherwise we risk mixing up packages of different versions or variants, though, with the fast working tree optimization, as long as the build system cooperates, we can ensure that we never mix up potentially colliding **packages** together during the execution of the build system.

For example, if we get a filesystem access for `lib/libboost_random.so`, you will be presented with choices for all Boost libraries available in nixpkgs, i.e. let's say version 1.81.0 and 1.80.0. If 1.81.0 is selected and you get another filesystem access for `lib/libboost_nowide.so`, because it would have been cached in the fast working tree, the **BuildFS** should not intervene here and you will get the 1.81.0 version of that library, without the fast working tree, **BuildFS** would have been consulted again for that library and another choice would have to happen which could lead to mixing libraries versions.

Nevertheless, the previous optimization cannot prevent the build system to request legacy headers that are only provided by legacy packages leading to a new risk of mixup, e.g. asking for the C header `xlocale.h` removed from glibc 2.26, still available in the bionic (Android) libc for example.

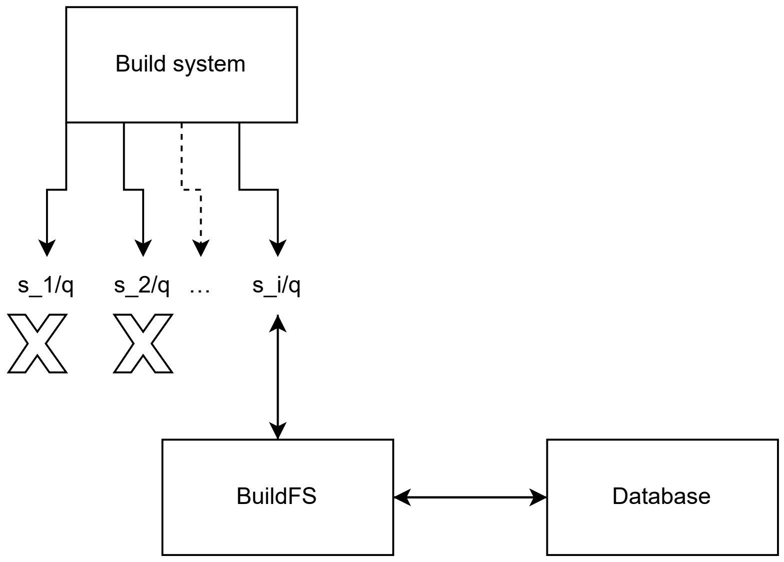

Figure 2: Dispatch high-level overview relative to s_i search paths

5.3 Toolchain and cross-compilation

Now that we have solid fundamentals to service build system requests, we need to take a look to compilation toolchains and cross-compilation contexts.

Given the problems we mentioned about C/C++ in section 1.2, there is one thing we cannot completely detect and for which we need special support in our custom filesystem, that is: the toolchain's specific files.

It is possible to install multiple versions of a toolchain on the same system, e.g. GCC and make the default C includes co-exist with the others via version namespacing: `/usr/include/gcc/v5.6.0/...`.

Though, in our case, our system has no concept of version and it is an undesirable property because, in a given project, system-level dependencies should never co-exist in different versions ideally.

That makes it even harder given that build system will perform capability detection based on the action of compiling small snippet C programs and testing if they compile or behave as intended, this information is difficult to intercept and analyze in a structured way.

Therefore, we put an assumption which comes out of our implementation experience: we assume that provided toolchains to the project are "self-contained" with respect to standard libraries and system libraries, this put a constraint on the provided package, some toolchains already use this mechanism under the name of "sysroot" sometimes to know where to look for their own files.

In our case, our nixpkgs-based implementation has wrapped toolchains who are aware of their own standard libraries with correct-by-construction hardcoded paths which are guaranteed to exist by virtue of Nix guaranteeing the presence of the build transitive closure inside the Nix store.

Going further, we addressed how to make normal toolchains behave correctly, though, cross-compilation is another concern when building packages. Some build systems may correctly ask for `$target_prefix-$toolchain`, e.g. `riscv64-linux-gcc`, which we call prefixed toolchain binaries.

For the same reasons, prefixed toolchain binaries may rely on different standard libraries and we want them to work out of the box as much as possible, hence, our choice to use self-contained toolchains pays itself again for cross-compiled variants of the toolchains.

5.4 Learn from user input: resolution recording

In the presentation, we mentioned the choice policy function Q, there are multiple ways to choose Q : popularity count on all packages, minimal number of paths among candidates, and each choice function can also be aggregated to form hybrid policies.

In our case, the fallback policy is the user choice, we will offer the user to choose among prefiltered candidates via the search query we presented in the previous sections and we will let them decide which package is the best for their usecase, that should covers all the cases, except for two.

When a package is missing from the prefiltered list, in which case, we could theoretically suggest to input a package by its name identifier. And, when a package is missing from the database, in which case, we could automatically run a packaging procedure which could lead to combine an automatic packager and buildxyz itself to source the native dependencies of that new dependency recursively.

As we accumulate information and knowledge on what are runtime patterns of a build system or a project, it is desirable to avoid having buildxyz re-ask for the same inputs as before.

Therefore, we introduce the concept of "resolution", that is, given a path q requested leading to prefiltered results, we will memoize these choices in a database of paths called the resolution database.

We build the resolution database into a tree of resolutions which can always be overridden by merging a new resolution database into it, e.g. considering resolution for `bin/python3` for Python 3.10 on the left and `bin/python3` for Python 3.11 on the right, Python 3.11 will win.

Furthermore, it is always possible to say that a path can be ignored, i.e. return `ENOENT` to a given path q, this is important to force feature detection to behave correctly.

Furthermore, the goal for BuildXYZ is to provide a seamless experience on class of software ecosystems, by analyzing this class and determining the minimal set of resolutions, it is possible to have automatic packaging without user interaction by relying on popularity count as a default policy.

We achieve this with a variant of resolutions called "core resolutions" which are resolutions always inserted in the seed database of BuildXYZ in the binary itself, maintained in BuildXYZ code, see for Python: https://github.com/RaitoBezarius/buildxyz/tree/main/data/python. In the case of Python, we will `ENOENT` any binary called `pgen` which is the old non-maintained and dropped parser generator for Python grammars which is still requested by Cython build system.

In the end, as application-specific package managers have a lock file to lock the inputs for their dependencies, our resolution recording is a lockfile for cross-language native dependencies.

Indeed, generating this lockfile in a pure mode would yield all the implicit data on which library is necessary, even the application-specific package manager would be part of it as it is a binary that is run in the environment, i.e. ‘pip‘ would be part of the lockfile, with its precise version, when installing a Python package with `pip`.

5.5 Learn from failures: parsing compilation errors and backtracking

In the previous section, we talked about the resolution recording which is currently done via user inputs, as said, this is not an inherent limitation, we will talk about possible interesting policies for automating resolution records.

The first trivial policy is to provide for the "best known match" according to some matching policy, e.g. "smallest size", "highest popularity in the graph of dependencies", etc.

The second policy is to perform backtracking on dependencies as soon as the build system fails, this is expensive on large dependency graphs, but, with existing resolutions, it can be used to automate massive resolution recording, more advanced system techniques can be applied like snapshotting a hierarchy of processes as checkpoints and rerun them to reduce the cost, achieving fuzzing-like techniques for packaging.

The third policy is to parse build system errors and infer what is the missing or corrective measure to take.

We implemented the first trivial policy and left the second and third policies as future improvements.

5.6 Protecting the user during operations

Package managers’ build system have different levels of expressiveness, for example, `setup.py` allowed users to express arbitrary code execution expressiveness via Python, Rust allows the same via `build.rs`. Nevertheless, few of them implement hardening measures to sandbox their arbitrary code execution process which poses a serious security risk.

On the other side, Nix chose from the get-go to make all builds go through a strict sandboxing process for various reasons, which, in turn, made certain class of vulnerabilities non-existent.

Providing a sandboxing measure for all BuildXYZ operations could improve the purity of the environment (and therefore the likelihood of reproducibility) and mitigate the security risks.

We provided a simple implementation relying on bubblewrap in BuildXYZ which has some caveats with some packages.

6 Empirical evaluation

Evaluation is driven on two dimensions:

- Ability to install **successfully** packages from a popular package index: PyPI, NPM, RubyGems
- Novel and previously-explored usecases

We also planned to measure on the performance overhead induced by running a build system under buildxyz, but due to a lack of time, this has not been done, though we believe this should not be a big problem because package management operations are dominated by network operations and compilation operations, while filesystem accesses could become a bottleneck, we can always invest into advanced FUSE performance features like the one described in [47] and [54].

We will first start describing `nixpkgs`, which is the package dataset we will use for `BuildXYZ`.

6.1 nixpkgs as a dataset

Nixpkgs is a large collection of packages for the Nix package manager.

As of May 2023, nixpkgs contains around 83,189 packages according to repology.org.

In terms of quality, nixpkgs is considered to be a high-quality database of packages due to its rigorous review process and continuous integration testing.

It is also known for its reproducibility and declarative approach to package management.

Overall, nixpkgs is a reliable and comprehensive resource to automatically provide dependencies through a FUSE filesystem.

6.2 Experimental setup

We use Nix semantics to describe as much as possible a pure and uniquely reproducible Linux environment, see table 3 for other external inputs. This is especially important as the evaluation should relies on a reasonably pure environment to not reuse dependencies external to the test harness, skewing the evaluation.

For evaluation over popular package indexes, we provide the data we used, we also describe how we built that data:

- PyPI: https://pypistats.org/top, preparation script: `contrib/prepare_pypi_dataset.sh`
  Running script: `contrib/evaluation/unsandboxed/iterate_over_pypi_dump.sh`
- RubyGems: https://rubygems.org/pages/data, import in PostgreSQL and a SQL query provided in the repository to extract the top 5000 Gems.

In the future, NPM is planned to be evaluated, but due to the difficult to access the data, we will use another proxy for popularity than PyPI and RubyGems and look at a form of popularity induced by the graph of dependencies in NPM.

We run each build system script in a pure Nix shell, i.e. most of the environment variables are cleaned up, we do keep the following environment variables described in table 4.

To the best of our knowledge, these environment variables are the most important ones responsible for an increase of impurity during evaluation because they are inherently hard to control.

6.3 Install evaluation

We consider taking the top 5000 of the PyPI Python index, NPM JavaScript index, RubyGems Ruby index.

Package registry	Success rate	Total packages
PyPI (Python)	50 %	4950
RubyGems (Ruby)	95 %	4927

Figure 3: BuildXYZ success rate in different package registries

The final total can be slightly different because of the testbed shutting down before the end of the operations.

6.3.1 Remarks on results

This evaluation is incomplete for PyPI due to the fact that nixpkgs was frozen at the start of this research (the mentioned revision refers to a commit made on the 24 March 2023), by the time, the evaluation has run: some PyPI packages already started using newest features for their latest versions which were not available in the compilers offered by nixpkgs.

For example, `pydantic` 2 shipped with (by then) nightly Rust features which were stabilized in a Rust compiler provided after the end of March 2023, therefore, when `rustc` is provided, compilation fails due to a nightly feature being used without marking it as a metadata, generally, this would not prove to be a big deal if it was possible to ignore this error and force Rust to proceed, unfortunately, not even the `RUST_BOOTSTRAP` environment variable can force the usage of nightly features on a stable Rust compiler if they are not marked accordingly.

Furthermore, we believe there are still more low-hanging fruits to extract to improve the accuracy score similar to the RubyGems one.

Finally, on the PyPI case, a lot of failures were difficult to reproduce on a local machine and surfaced only under parallelization of the testbed instead of running serially all tests, the actual score may be much higher, but will nevertheless make all reverse dependencies of failing packages like pydantic, fail in all cases.

Regarding the RubyGems score, there are two potential rationales for such a high score, either, this is because Ruby's package manager can have a bug and force the downloading of precompiled binaries which happens to be considered as working even in a pure shell without a filesystem hierarchy standard [41], either, this is indeed due to a better ecosystem regarding the compilation process of native dependencies, making buildxyz perform better on this specific set of packages.

In all cases, the author is planning to cleanup the evaluation testbed scripts and develop a reasonable continuous integration to monitor the evaluation score evolution across time, including automatic nixpkgs index upgrade, and extend the amount of registry to test.

6.4 Case studies

6.4.1 Automatic packaging

`BuildXYZ` can inject dependencies in the running environment, but they can also be recorded and replayed, therefore, it can be used as an API in other programs to provide fine-grained automatic detection of dependencies by ensuring the database has mapping with a distribution set of packages.

More precisely, in the case of the NixOS Linux distribution, the project `nix-init` [15] will take a project URL as an input and determine the corresponding Nix expression, it reuses the existing application-level package manager files to determine application-level dependencies, but cannot determine system-level dependencies.

A solution could consist of packaging it with application-level dependencies only, then run it in the Nix sandbox with `BuildXYZ`, recording the dependencies and augmenting the original Nix expression with the newly discovered system-level dependencies.

This usecase is not limited to Nix and could be applied to any other Linux distribution like Debian by mapping `nixpkgs` onto the Debian repositories or just replacing the `BuildXYZ` database by a Debian's one.

Input	Revision or value
BuildXYZ revision	`d0bc6a34219be467f5247b59975aaa44c7ee13a9`
BuildXYZ compilation mode	release
nixpkgs revision	`994e2ef9e9c70b4dd7257f73452a94e871723685`
Evaluation machine CPU	Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz
Evaluation machine RAM	128GiB
Evaluation machine temporary disk	1TB of persistent memory

Table 3: Evaluation test setup

Impure environment variables kept during evaluation		
Environment variable	Purpose	Default value
`BUILDXYZ_NIXPKGS`	Nixpkgs tree location	Defined by `flake.nix`
`RUST_BACKTRACE`	Debugging	unset
`MANUAL`	Automatic or manual packaging (debugging)	automatic packaging
`ENABLE_STRACE`	Run BuildXYZ under strace	disabled
`NIX_DEBUG`	C/C++ toolchain wrappers debugging	unset
`NIX_BUILD_CORES`	Amount of cores to use for build	dependent on Nix settings
`TEMPDIR/TEMP`	Temporary directory prefix	`/run/user/$UID`
`XDG_RUNTIME_DIR`	XDG runtime directory	`/run/user/$UID`
`NIX_BUILD_TOP`	Nix build top-level directories	`/run/user/$UID`
`system`	Host system	`x86_64-linux` for evaluation
`HOME`	Currently running user home directory	`/home/raito` for evaluation
`NIX_SSL_CERT_FILE`	SSL certificate bundle for HTTPS requests	NSS cacert 3.90's `ca-bundle.crt` for evaluation
`NIX_STORE`	Location of the Nix store	`/nix/store`
`TERM`	Terminal information	`tmux` for evaluation

Table 4: Environment variables preserved during the evaluation

6.4.2 Lazy package management

Lazy package management is the act of getting the relevant binaries in the environment only when you need them : sometimes when you run them.

`BuildXYZ` can be used as a lazy package manager to run binaries you do not have on your system.

For example, in a project, you can use one-off binaries by running `buildxyz binary ...`, the advantage compared to existing alternatives to lazy package management is that `BuildXYZ` is actually recursive in the dependencies, i.e. if that particular binary has another dependency, it will be brought during execution.

6.4.3 Diagnostics of filesystem patterns

As `BuildXYZ` listens carefully the filesystem accesses in fallback locations, it is a very interesting tool to learn and diagnose the behavior of a program in a blackbox or greybox fashion. Obviously, `strace` and similar tooling are better tailored for those usecases, but they do not focus on packaging-related filesystem accesses, `BuildXYZ` offers a simpler alternative to this usecase.

For example, the recent versions of the `pip` Python's package manager always asked for a `rustc` compiler multiple times in the same invocation, even if the underlying dependency required no Rust compiler.

The answer was that `pip` when sending a request to a registry computes a dynamic `User-Agent` depending on the availability of certain local programs on the system, in this case, if `rustc` was available in the context, a version string of that compiler was sent in the `User-Agent`, this simple feature was responsible for sending many filesystem access because no **cached** feature detection was performed as part of the initialization of this package manager.

Another example can be found in the GCC compiler toolchain, whenever compiling, it will attempt to find a `strip` binary in the context, but before that, it will ask for `gstrip`, which is the GNU version of `strip`, on (GNU/)Linux systems, `gstrip` is often absent, whereas on BSD systems, `strip` refers to the BSD strip program which behaves differently. Given that GCC will try to invoke the GNU versions of a binary before the unprefixed version all the time, it begs the question of: "can the cost of missed filesystem accesses be evaluated in build farms system?"

6.4.4 Run binaries without knowing the runtime dependencies

As we mentioned in 3.4, `LD_LIBRARY_PATH` cannot be served via the **BuildFS** as the current semantics of the dynamic loader search paths order are not in our favor.

Instead of patching the dynamic loader, we will take a step back and analyze which are the sets of binaries that requires to have `LD_LIBRARY_PATH`.

On NixOS, binaries are produced with all their libraries correctly set to an absolute path pointing to the Nix store, therefore, beyond dynamic `dlopen` calls, i.e. runtime dependencies, there is no need to override this environment variable. NixOS' binaries and NixOS' produced binaries are already correctly set from that perspective.

As `BuildXYZ` relies on nixpkgs, it will also produce binaries of the same form, as long as the runtime dependencies are correctly set, there is no problem.

Nevertheless, some programs may download binaries from Internet with a non-nixpkgs provenance, those will expect to have a dynamic loader at `/lib64/ld-linux-x86_64.so.2` for glibc based `x86_64` systems.

With `BuildXYZ` on non-NixOS systems, they will use the host Linux distribution dynamic loader at that location which will in turn use the filesystem hierarchy standard to find about those libraries. As we cannot control this dynamic loader, we will also consider this usecase out of scope.

Though, for NixOS systems, the same can be said except there is no default dynamic loader, therefore, any run will fail directly. A solution can be provided via a shim dynamic loader, the project `nix-ld` provides one which offers `NIX_LD_LIBRARY_PATH` in order to provide search path locations.

Assembling all of this in `BuildXYZ`, i.e. relying on a nix-ld being present and using `NIX_LD_LIBRARY_PATH` enable a user to run unpatched binaries without knowing the runtime dependencies as they will hit a controlled `BuildXYZ` location which will trigger the automatic runtime dependency dispensing.

This is what we believe to be a novel usecase because currently `nix-ld` relies on the fact you need to know about the runtime dependencies yourself, discover them and insert them in the environment, `BuildXYZ` composes well with that need to avoid manual discovery.

6.5 Real-world deployment experience

During its development, `BuildXYZ` already received interest from the community in two major applications:

- Integration with the automatic packager `nix-init` [15] 6.4.1
- Lazy package management 6.4.2 inside Docker containers

We already went through the first usecase in the previous sections, we will focus on the second usecase.

Usually, Docker containers can be built in various ways: importing a tarball, building a `Dockerfile`, Nix has an excellent support for the first, but requires your packages to be available in the Nix world, e.g. as derivations.

By starting from a base image containing the Nix package manager, it is still possible to assemble a `Dockerfile` in an imperative fashion while mixing outside of Nix packages and inside of Nix packages.

BuildXYZ can be used also to optimize the workflow to perform lazy package management and download only what is needed based on the project build system while leveraging nixpkgs.

7 Related Work

We will detail here the literature around our work and provide extra context around our original problem motivations.

7.1 On-demand dependency dispensing

On-demand dependency dispensing was an idea introduced in 2011 in [19] through a system call interposition technique.

Following this, on-demand dependency dispensing relies on a mechanism to provide on-demand the dependency: [23] computes the transitive closure of the required dependencies and build a final compressed archive which contains everything that is needed, it will use system call interposition to determine what is needed for the archive creation and re-execution via PRoot [49], a portable system call interposition engine. The archive content will usually contain a partial copy of the original filesystem. CARE also focuses on replicability across kernel versions, which is out of scope for BuildXYZ.

Then, in [21], the idea is to be able to run code snippets without manually installating the dependencies and evaluate a tool that can generate automatically `Dockerfile` based on a code snippet resulting in a Docker image containing the dependencies for a snippet. It achieves this by having explicit "language support" and parsers to visit the target's language AST and produce the required dependencies. For search, it relies on "database strategies", two are implemented: pip and APT and will look for exact matches over the network. Finally, it requires a Neo4j database seeded with a provided dump to perform dependency resolution.

7.2 Language-specific package managers

7.2.1 The case for C/C++

As we touch in 1.2 regarding C/C++ package managers, we are aware of [7] which is a C/C++ package manager used in some projects, but not standardized, see [32] for an account of their usage. Research also explored the usage of third-party libraries in C/C++ ecosystems in [43].

A module system which would naturally propose a package manager has been discussed in 2014 in [12] which unfortunately has not been adopted yet by C++ standardization committees.

Finally, in general, [24] explores the evolution and structure of package dependency networks but only on the language-level, through the resolutions generated by our work, it is possible to explore the "fully extended package dependency network" across system boundaries taking into account native cross-language dependencies of any ecosystem.

7.2.2 The case for Python

As we touch in 2.2, Python ecosystem is very interesting for scientific computing as acknowledge in [37], nevertheless, it suffers from a complexity problem as showed in [51] and corroborated by [6] analysis on growth of PyPI.

Furthermore, as wheels are prevalent, security concerns are difficult to control or tame, for example, in [50], an account of typosquatting attacks are studied, in a binary distribution model, we argue they would be harder to catch and an easy target to abuse given there is no real way to check that a precompiled wheel sent to the binary cache is the one we would expect, this can be compared to the recent controversy in Rust ecosystem with shipping precompiled macros, i.e. Rust binaries, in the popular deserialization library serde [46], which led to a new "pre-RFC" to support precompiled macros with a sandbox runtime via WebAssembly and a mechanism to control reproducibility of the precompiled file [44].

Our model avoids typosquatting by relying on nixpkgs which is not a self-service package registry where anyone can push any package name, though, we are still vulnerable to upstream supply chain attacks except if they happen at build-time where Nix sandbox can protect the user.

7.3 Version constraints solving

Related to the subject of **detecting dependencies**, the next one is **choosing the right version**, usually, this is performed by declaring version constraints and solving them.

In this ecosystem, there were also many ad-hoc solutions and there are still many ad-hoc solutions, but works like [45] explored a standardization of version constraints in a comprehensive way, taking into account, things like user preferences (e.g. minimize dependencies) in case of multiple matches.

Research explored the consequences of this work in [3], [1], [2].

Currently, only APT and OCaml's opam uses the CuDF format for version constraints solving.

7.4 Automatic repair based on build breakage data

We did not implement in this work automatic procedures to learn from build breakage data and generate potential hints or resolutions back and execute a form of backtracking to improve the automatic packaging procedure.

But this concept has been explored in multiple publications:

For Makefile-based mechanisms: [25] performs fault localization in Makefile, [4] conducts an empirical study on unspecified dependencies (what we also call "implicit dependencies" in this work) but only to the Makefile-level, [52] aim to rebuild a "unified dependency graph" composed also of the dynamic dependencies generated during the build and exploit it, the "UDGs" concept has also been used in [14] is a similar work.

Another set of works focus on the Java ecosystem and Gradle in [28] which will provide auto repair based on historical data, while [48] tries to understand build failures by analyzing logs and providing better hints to the developer, finally [30] will provide automatic repair in Maven via a build repair plan and a set of strategies like version update, dependency delete or add repository.

While most work seems to be skewed towards Makefile or Java, there is at least one work on automatic repair in the Python ecosystem aimed at improved reproducibility: [36] based on an error log analyzer and an iterative dependency solver.

Finally, some works focus on building a empirical study of compiler errors in various contexts such as continuous integration like in [29], [53], some work like [18] focus also on denoising the build breakage data.

7.5 Misc

This work heavily relies on Nix [10], a functional system-level package manager and was conducted on NixOS [11], a functional Linux distribution based on Nix.

While Nix was chosen for this work as it offers direct access to Nixpkgs, the model is generic enough to be usable with any similar package manager.

Two alternatives are possible:

- [8] which offers a Guile Scheme as a functional language for the expressions and its own package set adhering strictly to the Free Software Foundation policies.
- [16] which offers a Python domain specific language to configure packages and multiple versions of them.

Both alternatives share history with the Nix project, sometimes, even sharing pieces of code of the Nix package manager, e.g. the Nix store, for some time.

Functional-based systems like those differs from simply immutable operating systems like Fedora Silverblue or openSUSE microOS in the sense they are not functionally-oriented, therefore, they could not be used directly for this work as they adhere to the filesystem hierarchy standard [41], which gets in our way for this work.

8 Conclusion & Future works

We showed `BuildXYZ`, a system to automatically and recursively dispense system-level dependencies in an environment relying on Nix and FUSE.

It achieves successfully novel usecases around software packaging and refines already previously explored ones by using the largest software collection according to Repology, at the time of writing: `nixpkgs`.

`BuildXYZ` has solid foundations to achieve more fully automated behaviors while improving the output of resolutions to further advance the packaging problems.

In this section, we will see what are the interesting ideas, in our opinion, that a tool like `BuildXYZ` opens up to the packaging research community.

8.1 Fixing the missing data via materialization

As we explained in the first sections, the current situation of packaging is suboptimal: each language-specific ecosystem end up re-inventing a new package manager for their own semantics and all of these new package managers share a substantial amount of features between each other.

By looking at the packaging problem as a recipe that must be provided by the application-level package author to invoke the right system-level programs to install the missing system-level dependencies, we producing recipe for each system-level package manager, which is not manageable for all package authors.

Instead of this, we suggest to fill the gap between system level and application level dependencies by producing the missing data of "what system level paths do I need to install this application level package?".

Through the resolution records of `BuildXYZ`, this data can be recorded in a highly pure environment (A pure Nix shell combined with a sandbox).

Once we have this data in a raw format, an interesting aspect of it is that it is possible to expand it into the transitive closure of system dependencies tailored for any Linux distribution or environment, in our case, all that is required is mapping Nix store paths to other Linux distribution packages.

We call this process re-materialization or materialization in reference to the fact that every application-specific instructions or recipes contains this information encoded in a domain specific language catering to their specific combination of tools: a specific Linux distribution, system-level package manager, meta build system, etc, therefore, is impossible to machine read and reuse in other contexts.

With `BuildXYZ`, (re)materialization re-derives the encoded data in a black box fashion as long as it fits into our build system semantics.

We created a GitHub repository which we plan to offer as a dataset containing resolution records during evaluation runs: [26] which constitutes a first step into making this materialization data available.

8.2 A common format for cross-language and native dependencies

Once this data is rendered, it is still in a crude way and will not address the problem on the long run as even `BuildXYZ` cannot capture fully the complexity of package ecosystems, though, it serves as a stepping stone to bring ecosystem efforts towards a common format and tooling for cross-language and native dependencies and their locking counterparts.

Here, we draw inspiration from CUDF [45] which was made for version constraints : a new format could be produced for this and generic resolvers/solvers can be adopted by any language-specific ecosystem to gain automatic support for any distribution that does support this new format, similarly as what is described in [3], [1], [2].

In turn, we would reduce the problem from "having recipe at the application package" to "consuming and manipulating a data format in system level package manager or application level package manager".

Building on the top of the previous section, re-materializing on a large scale the existing data, fine-tuning further tooling to cover a maximum amount of the cases can bring forth a compatibility layer, good enough so we can start transitioning to produce natively the new data.

Finally, we will mention the recent ongoing efforts in the draft PEP 725 [38] which will provide for support of native dependencies inside the Python ecosystem, which could be the target output for BuildXYZ intermediate resolutions files if it is paired with a PURL (package URLs) from nixpkgs data to some generic PURL scheme.

8.3 Stable ABI to interop language-specific package manager and system package manager

Nevertheless, as practice evolves, e.g. containers, meta build systems, it is important to have a space for experimentation and new ideas.

Following the ideas of Gobo Linux in [20] and [35], described as the Alien interface, bringing forth a stable ABI/API to provide interoperability between language-specific package manager and system-level package manager would help having better integrations, it could even lead to an extension of the UNIX model and be an integral part of system components.

Standard libraries could interact in a structured way with this data and operate package operations such as installs and removals.

Finally, we will mention that `pkgconfig`, heavily used in our work, can also be considered to some extent as a form of stable "foreign interface" for locating libraries, as this can be seen in https://discuss.python.org/t/pkgconfig-specification-as-an-alternative-to-ctypes-util-find-library.

References

[1] Pietro Abate et al. "A modular package manager architecture". In: *Information and Software Technology* 55.2 (2013), pp. 459–474.

[2] Pietro Abate et al. "Dependency solving is still hard, but we are getting better at it". In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE. 2020, pp. 547–551.

[3] Pietro Abate et al. “Dependency solving: a separate concern in component evolution management”. In: *Journal of Systems and Software* 85.10 (2012), pp. 2228–2240.

[4] Cor-Paul Bezemer et al. “An empirical study of unspecified dependencies in make-based build systems”. In: *Empirical Software Engineering* 22 (2017), pp. 3117–3148.

[5] Ashish Bijlani and Umakishore Ramachandran. “Extension Framework for File Systems in User space.” In: *USENIX Annual Technical Conference*. 2019, pp. 121–134.

[6] Ethan Bommarito and Michael Bommarito. “An empirical analysis of the python package index (pypi)”. In: *arXiv preprint arXiv:1907.11073* (2019).

[7] C Conan. *C++ package manager*. 2018.

[8] Ludovic Courtès. “Functional package management with guix”. In: *arXiv preprint arXiv:1305.4584* (2013).

[9] Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. “Formal Aspects of Free and Open Source Software Components: A Short Survey”. In: *Formal Methods for Components and Objects: 11th International Symposium, FMCO 2012, Bertinoro, Italy, September 24-28, 2012, Revised Lectures 11*. Springer. 2013, pp. 216–239.

[10] Eelco Dolstra. *The purely functional software deployment model*. Utrecht University, 2006.

[11] Eelco Dolstra and Andres Löh. “NixOS: A purely functional Linux distribution”. In: *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. 2008, pp. 367–378.

[12] Gabriel Dos Reis, Mark Hall, and Gor Nishanov. *A Module System for C++(Revision 4)*. Tech. rep. Tech. Rep, 2014.

[13] Frank C Eigler et al. “Architecture of systemtap: a Linux trace/probe tool”. In: *Red Hat Enterprise SystemTap Document* (2005), pp. 17–20.

[14] Gang Fan et al. “Escaping dependency hell: finding build dependency errors with the unified dependency graph”. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2020, pp. 463–474.

[15] figsoda. *nix-init*. URL: https://github.com/nix-community/nix-init.

[16] Todd Gamblin et al. “The Spack package manager: bringing order to HPC software chaos”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–12.

[17] Tal Garfinkel. “Traps and pitfalls: Practical problems in system call interposition based security tools”. In: *In Proc. Network and Distributed Systems Security Symposium*. 2003.

[18] Taher Ahmed Ghaleb et al. “Studying the impact of noises in build breakage data”. In: *IEEE Transactions on Software Engineering* 47.9 (2019), pp. 1998–2011.

[19] Philip J Guo and Dawson R Engler. “CDE: Using System Call Interposition to Automatically Create Portable Software Packages.” In: *USENIX Annual technical conference*. Vol. 21. 2011.

[20] Michael Homer. “An updated directory structure for Unix”. In: ().

[21] Eric Horton and Chris Parnin. “DockerizeMe: Automatic Inference of Environment Dependencies for Python Code Snippets”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 328–338. DOI: 10.1109/ICSE.2019.00047.

[22] Bart Jacob et al. “SystemTap: instrumenting the Linux kernel for analyzing performance and functional problems”. In: *IBM Redbook* 116 (2008).

[23] Yves Janin, Cédric Vincent, and Rémi Duraffort. “Care, the comprehensive archiver for reproducible execution”. In: *Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering*. 2014, pp. 1–7.

[24] Riivo Kikas et al. “Structure and evolution of package dependency networks”. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE. 2017, pp. 102–112.

[25] Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. “Fault Localization for Build Code Errors in Makefiles”. In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ICSE Companion 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 600–601. ISBN: 9781450327688. DOI: 10.1145/2591062.2591135. URL: https://doi.org/10.1145/2591062.2591135.

[26] Ryan Lahfa. *BuildXYZ TOML resolutions*. 2023. URL: https://github.com/RaitoBezarius/buildxyz-examples.

[27] Samuel Laurén, Sampsa Rauti, and Ville Leppänen. “A survey on application sandboxing techniques”. In: *Proceedings of the 18th International Conference on Computer Systems and Technologies*. 2017, pp. 141–148.

[28] Yiling Lou et al. “History-Driven Build Failure Fixing: How Far Are We?” In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 43–54. ISBN: 9781450362245. DOI: 10.1145/3293882.3330578. URL: https://doi.org/10.1145/3293882.3330578.

[29] Yiling Lou et al. “Understanding build issue resolution in practice: symptoms and fix patterns”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 617–628.

[30] Christian Macho, Shane McIntosh, and Martin Pinzger. “Automatically repairing dependency-related build breakage”. In: *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)*. IEEE. 2018, pp. 106–117.

[31] Guillaume Maudoux and Kim Mens. “Correct, efficient, and tailored: The future of build systems”. In: *IEEE Software* 35.2 (2018), pp. 32–37.

[32] André Miranda and João Pimentel. “On the use of package managers by the C++ open-source community”. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 2018, pp. 1483–1491.

[33] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. "Build systems à la carte: Theory and practice". In: *Journal of Functional Programming* 30 (2020), e11.

[34] Leonardo de Moura and Sebastian Ullrich. "The lean 4 theorem prover and programming language". In: *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*. Springer. 2021, pp. 625–635.

[35] Hisham Muhammad, Lucas C Villa Real, and Michael Homer. "Taxonomy of package management in programming languages and operating systems". In: *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*. 2019, pp. 60–66.

[36] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. "Fixing dependency errors for Python build reproducibility". In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2021, pp. 439–451.

[37] Fernando Perez, Brian E Granger, and John D Hunter. "Python: an ecosystem for scientific computing". In: *Computing in Science & Engineering* 13.2 (2010), pp. 13–21.

[38] Ralf Gommers Pradyun Gedam. *PEP 725 – Specifying external dependencies in pyproject.toml*. URL: https://peps.python.org/pep-0725/.

[39] Nicola Prezza. "On locating paths in compressed tries". In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2021, pp. 744–760.

[40] Zhilei Ren et al. "Root cause localization for unreproducible builds via causality analysis over system call tracing". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 527–538.

[41] Rusty Russell, Daniel Quinlan, and Christopher Yeoh. "Filesystem hierarchy standard". In: *V2* 3 (2004), p. 29.

[42] Richard P Spillane et al. "Rapid file system development using ptrace". In: *Proceedings of the 2007 workshop on Experimental computer science*. 2007, 22–es.

[43] Wei Tang et al. "Towards Understanding Third-party Library Dependency in C/C++ Ecosystem". In: *37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–12.

[44] David Tolnay. *Pre-RFC: Sandboxed, deterministic, reproducible, efficient Wasm compilation of proc macros*. URL: https://internals.rust-lang.org/t/pre-rfc-sandboxed-deterministic-reproducible-efficient-wasm-compilation-of-proc-macros.

[45] Ralf Treinen and Stefano Zacchiroli. "Common upgradeability description format (CUDF) 2.0". In: *The Mancoosi project (FP7)* 3 (2009).

[46] Fabio Valentini. $using\ serde_derivewithoutprecompiledbinary-issue$#2538. URL: https://github.com/serde-rs/serde/issues/2538.

[47] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. "To FUSE or Not to FUSE: Performance of User-Space File Systems." In: *FAST*. Vol. 17. 2017, pp. 59–72.

[48] Carmine Vassallo et al. "Un-Break My Build: Assisting Developers with Build Repair Hints". In: *Proceedings of the 26th Conference on Program Comprehension*. ICPC '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 41–51. ISBN: 9781450357142. DOI: 10.1145/3196321.3196350. URL: https://doi.org/10.1145/3196321.3196350.

[49] Cédric Vincent and Yves Janin. "Proot: a step forward for qemu user-mode". In: *1st international qemu users' forum*. Citeseer. 2011, p. 41.

[50] Duc-Ly Vu et al. "Typosquatting and combosquatting attacks on the python ecosystem". In: *2020 ieee european symposium on security and privacy workshops (euros&pw)*. IEEE. 2020, pp. 509–514.

[51] Ying Wang et al. "Watchman: Monitoring dependency conflicts for python library ecosystem". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 125–135.

[52] Rongxin Wu et al. "Accelerating Build Dependency Error Detection via Virtual Build". In: *37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–12.

[53] Chen Zhang et al. "A Large-Scale Empirical Study of Compiler Errors in Continuous Integration". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 176–187. ISBN: 9781450355728. DOI: 10.1145/3338906.3338917. URL: https://doi.org/10.1145/3338906.3338917.

[54] Yue Zhu et al. "Direct-fuse: Removing the middleman for high-performance fuse file system support". In: *Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers*. 2018, pp. 1–8.