IMPROVING DEPENDENCY MANAGEMENT VIA FORMAL SEMANTICS

DONALD PINCKNEY

ABSTRACT

Dependency management is a key part of the software development lifecycle. Choices made when managing dependencies impact both the efficiency of software development and final software quality. Today's package managers automate much of the dependency management work, but key challenges remain generally unsolved, such as aiding developers with dependency repair, optimizing over dependency solutions, and improving transparency of popular package managers' semantics.

I claim that software dependency specifications can be generalized and formalized across diverse solving algorithms and package ecosystems, and that doing so enables us to attack such challenges. To support this claim, we show three contributions: a) an empirical analysis of dependency use in the NPM ecosystem, which motivates questions regarding the design of NPM, b) an executable formal semantics of dependency solving (PacSolve) and an optimizing NPM solver (MaxNPM), and c) a system for efficient automated repair of Python dependencies, modeled using PacSolve.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

APT Advanced Package Tool, a package manager for Linux distributions such as Ubuntu and Debian.

AST Abstract Syntax Tree, a data structure that represents the syntactic structure of source code.

BUG UPDATE In the context of semantic versioning, an update which increments the last component of a version number, e.g. `1.2.3` to `1.2.4`. Also called a patch update.

CABAL A package manager for the Haskell programming language.

CARGO A package manager for the Rust programming language.

CPAN The Comprehensive Perl Archive Network, a repository of Perl software modules.

CRAN A system for retrieving R packages and their dependencies from the Comprehensive R Archive Network.

CVE Common Vulnerabilities and Exposures, a reference number for publicly known cybersecurity vulnerabilities.

DEPENDENCY A relationship between two or more software components, where one component specifies that it requires the other in order to function.

DOWNSTREAM In the context of dependency management, a package which uses some upstream package. Also called a consumer or dependent package.

DPKG Debian Package Manager, a system for managing packages in the Debian operating system.

DSL Domain-specific language, a programming language designed for a particular domain or problem area.

LLM Large-language model, a very large transformer-based model for predicting the next token of text.

Major update In the context of semantic versioning, an update which increments the first component of a version number, e.g. `1.2.3` to `2.0.0`.

Maven Apache Maven, a software project management, packge manager, and build tool for Java-based projects.

Max-SMT A variant of SMT in which solutions which are both satisfying and optimal for a particular objective function are found.

Minor update In the context of semantic versioning, an update which increments the middle component of a version number, e.g. `1.2.3` to `1.3.0`.

NPM A package manager and package ecosystem for the JavaScript programming language.

Package manager A software tool that manage dependencies between packages in a project, such as NPM and PIP.

PIP A package manager for the Python programming language.

Pyenv A tool for managing multiple Python versions and environments on a single system.

Semver Semantic versioning, a standard for expressing software versions as a triple of numbers that relate to the level of change in the software.

SMT Satisfiability Modulo Theories, a technique used to determine whether a given formula is satisfiable or not in a particular theory.

Spack A package manager specifically designed for building, testing, and installing packages in a reproducible and deterministic manner.

Unification A concept used in computer science and programming languages, referring to the process of combining multiple elements into a single, unified whole.

UPSTREAM In the context of dependency management, a package which is used by other downstream packages. Also called a dependency.

VENV A virtual environment manager for Python, used to isolate dependencies and manage package installations.

WEBPACK A JavaScript module bundler that allows developers to manage dependencies and build their applications for various environments.

YARN A package manager for JavaScript.

1 INTRODUCTION

Nearly all software written today relies on dependencies, such as depending on critical utility libraries (e.g. `openssl`), or architecting an entire application around a framework (e.g. `react`). These dependencies in turn rely on their own set of dependencies, forming a complex dependency graph.

Installing and managing dependencies (and transitive dependencies) manually is impractical for any large project. Versions of dependencies must be chosen carefully such that they are compatible, and updates must be likewise computed properly to ensure consistency. Additionally, every dependency affects the quality of the compiled software in multiple dimensions, including code size and security, with different versions of dependencies having different effects.

To automate dealing with these complexities, various communities have developed *package managers* to assist with dependency management, both at the system level (e.g. APT, Spack [35]) and at the language level (NPM, PIP, etc.). These package managers typically consist of several components:

1. a large ecosystem of open-source packages (over 500K in PyPI [31] and 3.4M in NPM [23] at the time of writing),
2. a syntax for writing *version numbers* of packages, and allowing multiple versions of a package to be published to the ecosystem,
3. a syntax for writing *dependency specifications*,
4. and a command-line tool, a *dependency solver*, which takes as input dependency specifications, computes a dependency solution, and performs the associated installation actions.

Modern package managers offer programmers several benefits that are worth remarking on. First, dependency specification syntaxes allow programmers to specify their dependencies in a declarative, typically constraint-based style, such as "any version of `react` greater than 1.1.0". Programmers are freed from both the effort of manually updating dependencies in response to e.g. security patches, as well as the pain of

writing imperative build scripts to properly compile or install dependencies. Second, among the large space of potential dependency solutions induced by the dependency constraints, most package managers try to find a solution that is "good" in some sense, such as trying to select newer versions of dependencies (Chapters 3 to 5 explore this empirically and formally).

However, there are also multiple shortcomings with many of these systems, which in this dissertation I show can be addressed by developing a thorough formalization of dependency solving, and using it to build better dependency management tools. Below I outline some of the current problems faced by programmers when managing dependencies.

ARBITRARY CHOICES IN THE DESIGN SPACE First, the precise semantics of the dependency specification languages are typically not formally defined but rather are considered to be defined by the implementation of the package manager. While the dependency specification languages are mostly self-evident in meaning, there are nevertheless various surprising behaviors or "footguns" [53] which indicate that a formal treatment of dependency specification would offer explanatory value. Similarly, the precise behavior and guarantees offered by the solving algorithm are likewise not always well understood. Programmers may already struggle to understand all the quirks of a single package manager, and this is compounded by subtle differences between package managers. Moreover, these package managers make trade-offs in the design space of dependency management, as some goals (choosing newer versions of packages) may be in tension with other goals (reducing the number of dependency conflicts), and unsurprisingly, different package managers bake different trade-offs into their design.

LACK OF SOLUTION EXPLORATION Second, even though package managers expose a declarative-style language for specifying dependencies, they do very little to exploit this structure. Dependency constraints typically define a large space of admissible dependency solutions, but most package managers do not explore this space thoroughly. Instead, they typically use simple heuristics to directly choose an appropriate solution, and only explore more of the space when needed to backtrack due to conflicts[1]. This leaves on the table opportunities for finding even better dependency solutions. Some package managers, such as Spack [35] and OPIUM [101], solve dependencies

1 NPM does not even do backtracking on conflicts.

by transforming dependency problems into some type of constraint programming problem, and solving that using a solver. This separation allows for easier expression of cost functions over the space, which both my work and prior work exploit.

DEFECTS IN DEPENDENCY SPECIFICATIONS Third, all package managers assume the validity of the human-provided dependency constraints and version numbers. These annotations are not checked against the underlying code, and thus can easily be published to the public package ecosystems while incorrect. Specifying dependency constraints which are both flexible and correct is inherently intractable, as programmers of dependencies can always publish updates which they claim to be compatible updates but actually are breaking changes. In practice, specifying dependency constraints is a difficult balance between writing constraints that are too narrow and too broad. A constraint that is too narrow (admits a strict subset of the versions that work with the program) does not cause bugs, but restricts the admitted solution space potentially causing solving failures or rejecting higher-quality solutions. On the other hand, a constraint that is too broad (admits versions that do not work with the program) is a serious bug, as it means that (depending on the exact solving context) the program may be linked with a dependency version which causes bugs or total program failure. Programmers of published packages have a responsibility of making their constraints neither too broad nor too narrow, a task which is made more difficult by the fact that they do not have complete information, since more versions of dependencies will be uploaded in the future after the programmer has specified the dependency constraint.

A partial solution to this problem is *semantic versioning*, or *semver*, which acts as an informal contract between package authors and package consumers, in which package authors promise to publish breaking changes as increments to the major version number, such as `1.2.5` to `2.0.0`. If package authors follow this convention perfectly, then consumers may feel comfortable writing dependency constraints which include all versions of the form `1.*.*`. As with dependency constraints, adherence to semver conventions is never validated. Defects in dependency specifications or version specifications lead to broken dependencies, necessitating the need for *dependency repair* techniques.

1.1 THESIS

Package managers have become essential tools for programmers across many ecosystems. While they have automated much of the grunt work of repetitive dependency management tasks such as solving constraints, downloading, building, and linking dependencies, key problems remain in the space of dependency management, including searching for optimal dependency solutions and repairing dependency solutions. Since package managers are used so widely in practice, we now have massive datasets of detailed dependency constraints for us to leverage and evaluate on.

In this dissertation, I present a collection of work that addresses many of these loose threads of dependency management. I claim that:

> Software dependency specification and management can be formalized, and doing so provides a basis for building tools offering improved optimization and repair of dependencies.

Concretely, four contributions form the support for this thesis:

1. An empirical analysis of how programmers tend to use dependencies in practice, as well as how dependency solving behaves at scale.

   This empirical analysis is discussed in Chapter 3, which is based on the MSR 2023 paper "A Large Scale Analysis of Semantic Versioning in NPM" [80].

2. A formal semantics of dependency solving (PacSolve).

   This formal semantics is presented in Chapter 4, and is an extended version of the semantics developed in the ICSE 2023 paper "Flexible and Optimal Dependency Management via Max-SMT" [84].

3. A drop-in optimizing solver replacement for NPM (MaxNPM) built using PacSolve.

   The MaxNPM tool and empirical evalutions are presented in Chapter 5, which is based on the above-mentioned ICSE 2023 paper.

4. A system for automated repair of Python dependencies (Repyro), modeled using PacSolve.

   This work is discussed in Chapter 6.

The first two contributions form a basis for understanding dependency management, and the latter two build on that understanding to implement novel tools.

2 BACKGROUND

Dependency management and package managers are currently bifurcated between "system package managers" (e.g. APT [42] for Debian Linux), which install software onto a local machine, and "programming language package managers" (e.g. NPM for JavaScript) which install libraries written in and for that specific language. While the technical contents of my thesis have primarily been applied to programming language package managers, they are equally applicable to system package managers, and moreover serve to highlight that system and programming language package managers are not very different in essence. We start the story of package management by looking at the distinct but related topic of module systems.

2.1 MODULE SYSTEMS

Since the early days of computing, mechanisms for program abstraction (broadly defined) have been a crucial object of study in programming languages research. Various such mechanisms have been developed, including the concepts of abstract data types [59], classes [76], and most relevantly to this thesis, modules, introduced by Wirth in Modula [105], and expanded upon by MacQueen and others surrounding the development of ML [43, 44, 56, 62, 63, 69].

While module systems across languages vary substantially, at their simplest they provide programmers three key features: a) writing definitions (classes, functions, types, etc.) scoped inside a module, b) *exporting* definitions from a module, making them accessibly by other modules, and c) *importing* definitions from some other module into the current module, making those external definitions locally accessible. These essential features of module systems allow programmers to decompose their software into mostly-independent modules, thereby promoting better software design and enabling programmers to comprehend modules in isolation from each other.

Research on the design of modules systems has focused on both design quality questions (expressiveness, ease-of-use, support for separate compilation, etc.) as well as formal questions of the soundness of type checking across modules. However, in this

pre-Internet time when researchers were first developing these module systems, it was difficult to transmit and share code with other collaborators, let alone strangers.

2.2 PACKAGE DISTRIBUTION SYSTEMS

Some of the earliest systems for sharing code across the Internet were born from users of specific programming languages and their need to collaborate and more easily reuse the work of others in the community. CTAN (the Comprehensive TEXArchive Network) [40], first discussed in 1991 and implemented in 1992, was the first centralized system for TeX users to be able to find and download packages (i.e. TEXsource code) submitted by other users, as well as submit packages of their own. Other archive networks such as CPAN (Perl) [98], CRAN (R) [99] and others soon followed. Concurrently, early Linux distributions realized the need for centralized software repositories paired with tooling (such as dpkg) to enable users to install software from these repositories which would be packaged in a well-defined and automatically installable format.

Early versions of these systems were solving the problem of *package distribution*, but did not (initially) think about *dependencies*. To install a package, a user would need to be aware of what its dependencies were (e.g. from documentation), use the package installation tools to install those dependencies, and then install the primary package they wanted. Updating packages to newer versions would similarly require updating dependencies first, and then updating the primary package. Another key difference between these package distribution systems and other modern package ecosystems (including PyPI, Crates, and NPM) is that they were (and still are) highly curated, requiring package authors to fill out detailed forms to register packages submissions which are subjected to manual review. Some challenges with dependency management, such as being able to trust that dependency updates really are non-breaking, are exacerbated in ecosystems at the scale of NPM for which manual review is not conducted.

2.3 DEPENDENCY SPECIFICATIONS AND PACKAGE MANAGERS

To enable significantly easier package installation, designers of both these operating system package managers (pms, rpm, apt, etc.) and programming langauge package managers (CPAN, Maven, etc.) designed formats for package authors to specify package

dependencies in a machine-readable format, which the package manager would read at install time and then use to install the required (transitive) dependencies.

While different package managers have wildly varying notions of dependency specifications and automated solving algorithms, these concepts, paired with centralized package repositories, are the essential components of what we would now call a "package manager". Chapter 4 will present a formalization of these components in greater detail, so for the moment we will informally assume that package authors write "dependency specifications" by specifying the name of the dependent package, and possibly some versioning information.

Package managers are used in a separate installation phase before the installed packages are loaded[1], and the package manager implementation is generally separate from the system which loads the packages. For example, multiple package managers exist for JavaScript (NPM, Yarn, etc.), all of which install packages on disk in a standard format (the `node_modules/` directory) which is then loaded by the module system of the Node runtime. Package managers thus act as a bridge between the package repository (e.g. `npmjs.com`) and the module system of the programming language in question. The package manager receives input from the user, looks up dependency data from the package repository, computes some sort of dependency solution, and finally translates that solution into a format appropriate for the language's module system. We will pursue a deeper discussion of modeling this formally in Chapter 4.

2.4 DEPENDENCY MANAGEMENT

Dependency *management* involves more than just one-off solving of dependencies. Programmers must determine policies for how to effectively evolve their dependencies over time, while balancing factors such as prioritizing up-to-date depdencies and minimizing risks that come with that, such as accidental breaking changes, malicious updates, code size bloat, handling dependency conflicts, and more. As open source package repositories grow in scale, the maintenance, updating, and distribution of packages is difficult and time-consuming for programmers to maintain. Of particular concern is the *technical lag* [15, 25, 39, 112, 114] that packages experience between when a new update is available for a dependency and when that update is applied.

1 "loaded" may mean executed (in the context of a system package manager providing a binary) or imported (in the context of a programming language package manager).

To ease this maintenance burden, many package managers support some form of (semi-)automated dependency updates. Package managers typically allow programmers to write *partially flexible* constraints on version numbers of dependencies. For example, a JavaScript programmer may specify that they depend on the package `react`, with constraint `^18.1.1`, which (in the semantics of NPM) means that updates are allowed until but excluding version `19.0.0`. Simultaneously, the publisher of `react` owns the responsiblity of incrementing the version number from `18.x.y` to `19.0.0` if and only if that update introduces client-breaking changes. This particular scheme is known as semantic versioning ("semver"), and is used widely by NPM and other similarly-designed ecosystems (PyPi, etc.) Other schemes exist, with variance in the syntax of both version numbers and version constraints. Generally though, flexible version constraint systems allow developers of dependent packages to specify which types of updates they are willing to automatically accept. Ideally, this helps developers to express constraints and version numbers so that non-breaking important updates (such as security patches) flow rapidly to downstream packages, while breaking changes are delayed until developers choose to accept them.

The effectiveness of these systems depends on both formal questions (what are the precise semantics of constraints, how are conflicts handled, etc.) and on human behavior (of both package clients and publishers). Modern package managers support a wide-array of version constraint syntax paired with complex solving algorithms, which, while expressive, presents opportunities for both technical/formal deficiencies and for human confusion/error. We start by exploring how programmers write version numbers and constraints in practice within the NPM system in Chapter 3, and then discuss a formal framework for dependency management in Chapter 4.

3 EMPIRICAL STUDY OF DEPENDENCY CONSTRAINTS

3.1 INTRODUCTION

Modern software development relies inextricably on open source package repositories on a massive scale. For example, the NPM repository contains over two million packages and serves tens of billions of downloads weekly, and practically every JavaScript application uses the NPM package manager to install packages from the NPM repository. Understanding the properties of the software supply chain is vital, as it determines if security patches and other updates will flow to downstream clients, and what attack opportunities bad actors may have for malicious updates.

As discussed in Chapter 2, one particular concern is the *technical lag* [15, 25, 39, 112, 114] between installed and available dependency versions, and one potential solution is semantic versioning ("semver"), in which versions are numbered in the form `major.minor.bug`, where `major` denotes breaking API changes, `minor` denotes a non-breaking change adding new functionality, and `bug` denotes a backwards-compatible bug fix[1][87].

However, there are three significant complications with semver in practice that can lead to technical lag [15, 25, 39, 112, 114]. First, the positive properties of semver are predicated on both upstream developers labeling their updates with the correct semver increment type, and on downstream developers using constraints that are neither too flexible nor too strict. Second, dependencies in the middle of a transitive dependency chain affect the final received versions of dependencies. The downstream developer may list a constraint that allows the most up-to-date version of a package, but if a transitive dependency has a more restrictive constraint, the downstream developer may not receive the up-to-date version. Third, allowing for automatic (bug) updates to dependencies can be dangerous, as it introduces an attack vector for malware.

In this chapter, we aim to understand how developers make use of dependencies, semantic versioning, and flexible version constraints at the ecosystem-scale, and how all these factors intersect to affect developer experience and supply chain security. Prior

1 I use the term "bug" rather than the standard "patch" semver terminology, so as to disambiguate from the notion of *security patches*.

work on mining data from the NPM ecosystem has primarily focused on answering questions about NPM at a snapshot in time [4, 16, 52, 111]. Here, we first understand how developers make use of semantic versioning by analyzing flexible constraint type frequency and semver increment type frequency over the entire history of NPM. Then, to understand how updates flow in practice at the ecosystem scale, we run large-scale experiments that resolve packages' dependencies at different snapshots in time, observing how long it takes for updates to be received by downstream packages. To enable these experiments, we built a tool that allows for accurate time-travel dependency solving throughout the history of NPM. This methodology allows for more precision in resolving dependencies throughout time compared to prior work [25, 26, 60, 113, 114] which approximated NPM's behavioral semantics, which are not well-specified [84].

In total, we have built the first dataset of NPM that includes (as of October 31, 2022):

1. every package on NPM (2,663,681 packages)
2. every version of every package (28,941,927 versions)
3. metadata ($\approx$ 40 GB compressed) and packaged code ($\approx$ 19 TB compressed) for every version of every package,
4. full data of security advisories issued for NPM packages, downloaded from the GitHub Security Advisory database.

This dataset is indexed to allow for easy querying and large-scale distributed computations. To gather this data, we designed and implemented a distributed system for downloading, archiving and retrieving packages from NPM. We release our scraper and dataset under the BSD 3-Clause license[2].

We use our dataset to answer several questions about the NPM ecosystem, in particular how developers use semantic versioning, and how this affects supply chain security:

- **RQ1**: Do developers specify dependency version constraints to allow for automated updates?
- **RQ2**: Do developers use semantic versioning in their package releases to allow for automated updates to downstream packages?

2 Please see https://dependencies.science for access to up-to-date metadata, tarball data, and source code. The original artifact excluding tarball data is available on Zenodo [82].

- **RQ3**: Do packages frequently contain out-of-date dependencies? And when updates are published, how long until those updates are received by downstream packages?
- **RQ4**: Among the types of semver updates, what types of high-level changes do developers tend to make? How often do developers only update dependencies?

These results are impactful for both developers and researchers. We show that, generally, the NPM ecosystem is effective in terms of efficient distribution of non-breaking updates, but most packages end up with out-of-date dependencies anyways due to the sheer volume of dependencies and updates to deal with. In addition, we found evidence that some developers use semver non-optimally when releasing security patches, and that minor and major semver updates appear to have a higher risk of introducing security vulnerabilities.

3.2 METHODOLOGY

At a high-level, we answer our four core research questions using different aspects of our dataset and analysis systems. RQ1 and RQ2 are answered purely via analysis of our scraped metadata. Answering RQ3 is more challenging as it requires reasoning about how dependencies are resolved across time, which we answer by using our time-traveling dependency resolver in large-scale experiments. Finally, to answer RQ4 we compute diffs between tarballs of package versions.

3.2.1 *RQ1: Version Constraint Usage*

Within NPM's rich language for specifying version constraints on dependencies [78, 84], it is unclear which of the many constraint types developers frequently make use of and how loose or restrictive those constraints are.

We classify version constraints in the following mutually exclusive categories:

1. Exact constraints (`"=1.2.3"`) accept no versions other than the specifically listed one;
2. Bug-flexible constraints (`"~1.2.3"`) accept any updates to the bug semver component, so `1.2.4`, etc.;

3. Minor-flexible constraints (`"^1.2.3"`) accept any updates to the minor semver component, so `1.3.0`, etc.;
4. Geq constraints (`">=1.2.3"`) accept any versions greater than or equal to the specified version;
5. Any constraints (`"*"`) accept any versions; and
6. Other constraints, such as disjunction, conjunction, GitHub URLs, etc.

We then examine frequencies of these constraint categories across NPM, segmented by year so we can observe how constraint usage has evolved historically. In addition, one challenge with analyzing data from NPM is that some packages publish a massive number of versions (React has over 1,000 versions), so aggregating across all versions may produce results that are biased towards packages with more versions. In RQ1 we select only the most recent version of every package that was uploaded within each year. This enables us to segment by time while avoiding this bias.

3.2.2 *RQ2: Semantic Versioning in Updates*

We now turn to examine how developers increment their semantic version numbers when publishing updates. We first find all of the package updates that have occurred in NPM's history, and classify each as a bug (e.g. `5.4.8` $\rightarrow$ `5.4.9`), minor (e.g. `5.4.8` $\rightarrow$ `5.5.0`), or major (e.g. `5.4.8` $\rightarrow$ `6.0.0`) update.

One would expect that updates can trivially be identified as consecutive versions of the same package. NPM however allows versions to be published non-chronologically. This feature allows for maintenance of parallel version branches. For example, consider the following *chronological* order of versions: `1.0.0`, then `2.0.0`, then `1.0.1`, and then `2.0.1`. In this example, the mined updates should consist of: `1.0.0` $\rightarrow$ `2.0.0`, `1.0.0` $\rightarrow$ `1.0.1`, and `2.0.0` $\rightarrow$ `2.0.1`, as these reflect updates that are most closely based on the source version while being chronologically and numerically consistent. We would not include the update `1.0.1` $\rightarrow$ `2.0.0` because it is not chronologically consistent, and thus `2.0.0` is unlikely to be a derivative of `1.0.1`.

To determine the set of updates, we group versions by the equivalence relation of same major component and assert that groups are ordered within themselves chronologically. We then have updates between versions within each group, and between different groups. Continuing the above example, we have two groups: {`1.0.0`, `1.0.1`}

and {`2.0.0`, `2.0.1`}. From intra-group ordering we obtain `1.0.0` → `1.0.1` and `2.0.0` → `2.0.1`, and from the inter-group ordering we obtain `1.0.0` → `2.0.0`. We believe this algorithm reflects well how developers publish updates, and we discuss alternatives in Section 3.6. When computing these updates, we first filter out all prerelease versions (e.g. `1.2.3-beta5`), yielding 1,453,789 packages with at least one update (of 2,869,085 packages). We then filter out 52,279 packages that do not have consistent intra-group chronological orders.

With all updates and version increment types identified, we examine the distribution of the three update types across the whole population, and then compare to the subgroups of updates that introduce and patch vulnerabilities. Updates that patch vulnerabilities are identified directly in the scraped advisory database, while we identify versions that introduce vulnerabilities as the minimal version containing that vulnerability. To avoid the bias introduced by some packages having a large number of updates, our top-level aggregation is among packages rather than updates. For each package, we identify the proportion of its updates of each type (segmenting by security effect), and then visualize this percentage across all the packages. This enables us to make conclusions about how packages and package developers generally handle incrementing semver numbers during updates. In addition, note that when segmenting by updates that introduce vulnerabilities, we are *not* attempting to study malware, rather updates that (probably inadvertently) introduce a vulnerability.

3.2.3 *RQ3: Out-of-Date Dependencies and Update Flows*

The properties examined thus far have been local properties of each package, in that each package has been analyzed individually. We now wish to answer how out-of-date NPM packages typically are, and how long it takes updates to flow to downstream packages. Both of these properties rely on all the packages in the transitive dependency closure of a downstream package. However, reasoning precisely about how dependencies are solved is challenging both because NPM's dependency solving algorithm is complex (Section 7.1.2), and because we wish to parameterize this over time.

In order to compute solutions accurately and at different points in time, we use NPM's solver combined with a proxy that emulates the world state at any given point in history (described in Section 3.3.3). With this key tool, we then perform two experiments: first we solve the dependencies of the most recent version of every package in NPM and observe how many packages have out-of-date dependencies; we then

explore how updates flow to downstream packages by solving the dependencies of the downstream package at different points in time until it receives the update.

3.2.4 *RQ4: Analyzing Code Changes in Updates*

After having examined how developers use constraints and version numbers in isolation, we next align that with a high-level characterization of what updates actually change. For every identified update, we decompress the packaged code from both versions, and look for file changes. We then classify changes as modifying dependencies (in the `package.json` file), code (`.js, .ts, .jsx, .tsx`), both, or neither (such as only modifying configuration files or updating a README file). We then examine the distribution of these types of changes segmented by semver increment type, again normalizing per-package to avoid biasing towards packages with more updates.

Analyzing at a deeper level is possible with our dataset, but is beyond the scope of this work. Note that many packages upload compiled or minified JavaScript code, which makes it difficult to even look at simple line-by-line diffs. In addition, we could have chosen to count other file types as code (`.sh`, etc.), but we chose to focus on JavaScript code.

3.3 SYSTEM ARCHITECTURE

In order to perform our methodology, we needed a system that could scrape and store all metadata and tarball data, and allow us to perform analyses and experiments on both the metadata and tarball data. This system needs to be able to run on our academic Slurm-backed [92, 110] HPC cluster. To solve this problem, we designed our own system, organized into 3 primary components (Fig. 3.1):

1. The Metadata Manager, which continually scrapes data from NPM and periodically from the GitHub Security Advisory Database;
2. the Job Manager, which receives job requests (either tarball download jobs or parallelizable compute jobs) and then coordinates job execution and distributed file system locks; and
3. the Compute Cluster, in which we can spawn worker nodes and access a networked file system.

We now explain how we accomplish the primary tasks required by our methodology.

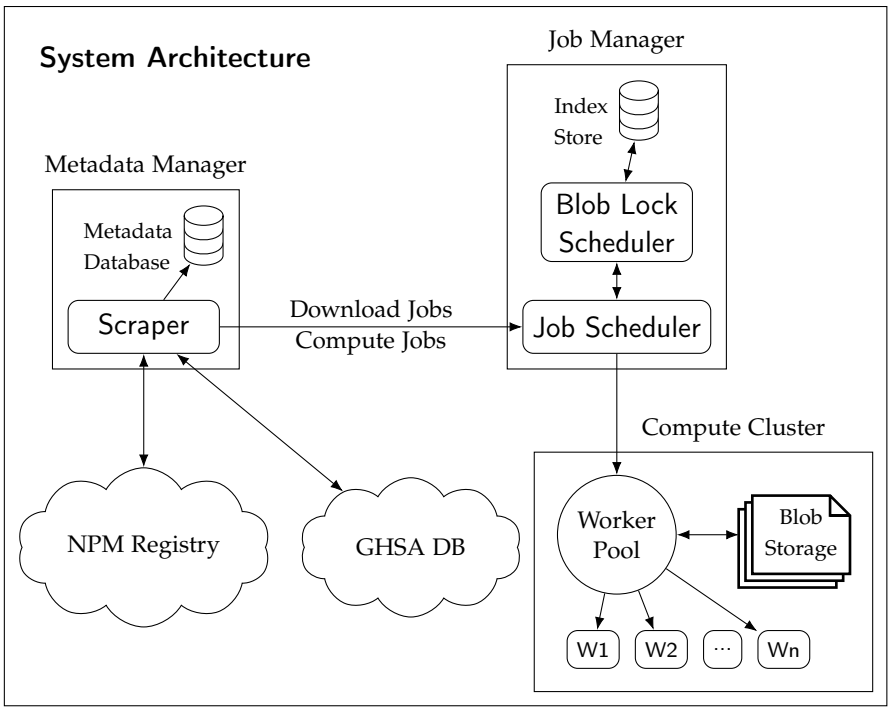


Figure 3.1: Overview of our system architecture.

3.3.1 *Metadata Acquisition*

NPM stores metadata in a CouchDB database. CouchDB is a document-oriented JSON database and is a good fit for NPM because it is schemaless and allows for arbitrary nesting of JSON objects, such as the `package.json` file. For performing data analysis we find it to be a poor fit due to the extremely loose structure. There is almost no validation of the `package.json` files in the CouchDB, making it difficult to use for analyses without first cleaning the data (normalizing date formats, normalizing dependency formats, etc.).

The Metadata Manager (top left of Fig. 3.1) continually receives metadata changes from NPM via their changes API [75], validates those changes, and inserts the data into PostgreSQL [41] (Metadata Database in Fig. 3.1). Additionally, the Metadata Manager periodically scrapes the GitHub Security Advisory Database (GHSA DB in Fig. 3.1) and

imports the security metadata into PostgreSQL as well. RQ1 and RQ2 can be answered entirely via issuing PostgreSQL queries to the Metadata Manager.

When metadata changes are received that contain URLs to new package tarballs, the Metadata Manager enqueues a tarball download job to then be handled by the Job Manager.

3.3.2 *Tarball Data Acquisition and Compute Cluster*

For scraping and storing package tarballs, we need to be able to store tens of millions of tarballs, while allowing for both concurrent writes to the storage since new tarballs are downloaded continually, as well as concurrent reads from the storage when performing analyses.

The worker nodes within the Compute Cluster are connected via a networked file system. One interesting approach would be to use a technology such as Hadoop [32] on top of the networked file system to accomplish this. However, we did not explore this approach out of concern of Hadoop's scalability with regards to storing many small files [6] (our use case). In addition, we are are unsure if Hadoop can run correctly and efficiently on top of a networked file system.

Instead, we store tarball data in a custom-built blob storage system stored on the networked file system (bottom right of Fig. 3.1). The Job Manager (top right of Fig. 3.1) controls access to the blob storage, keeping track of byte offsets and coordinating locks for writing, while individual worker nodes in the Compute Cluster perform the networked disk I/O.

Tarballs are downloaded when the Job Manager receives a tarball download job request from the Metadata Manager, at which point it assigns the download job to a single worker node. The Metadata Manager also offers an compute API which can receive arbitrary Rust code to run as a parallel map operation across all packages. The Job Manager handles compute job requests by distributing the compute task across many worker nodes, compiling the Rust code on-demand, and optionally allowing each node to perform lockless read-only operations from the blob storage.

This system allows us to continually scrape and store tens of millions of tarballs, and to efficiently retrieve them for computation when answering RQ4. Additionally, while RQ3 does not read from the blob storage, it follows the same compute workflow.

3.3.3 *Time-Traveling Dependency Resolver*

In order to carry out our experiments outlined in Section 3.2 for RQ3, we needed to be able to observe how a package's dependencies would have been solved at arbitrary points in NPM's history. We built a proxy server that can be used with vanilla NPM to enable time-travel dependency resolving.

NPM's command line tool enables the user to specify a custom package registry to use in place of `npmjs.com`. To use our time-traveling resolver, we specify a registry base URL pointing to our proxy server that includes in the URL the timestamp to time-travel to. The proxy server then receives the timestamp and can then rewrite responses from `npmjs.com` to remove versions of packages after the timestamp. Since this does not rely on the rest of our system, it is extremely easy to setup and use. However, in order to scale the computation across the dataset, we use the compute capabilities discussed above in Section 3.3.2.

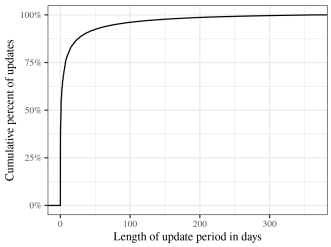
3.4 RESULTS

At a high level, we would consider a package ecosystem to be healthy with regards to update distribution when updates that are positive (performance improvements, bug fixes, security patches, etc.) can be quickly and easily adopted by downstream dependencies, while disruptive changes (security vulnerabilities, malware, etc.) flow more slowly. In NPM, the flow of updates is determined by two factors: how do downstream developers tend to specify version constraints for dependencies (RQ1), and how do upstream developers tend to increment their version numbers when releasing updates (RQ2). We start by explaining the overall structure and general properties of the dataset. Then we move on to discuss RQ1 and RQ2 separately, and finally we consider how RQ1 and RQ2 intersect in practice in the ecosystem (RQ3), and how they are related to the actual contents of the updates (RQ4).

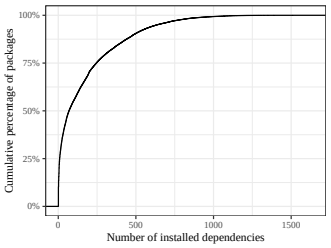
3.4.1 *Dataset Structure and General Properties*

As discussed in Section 3.1 our collected data is split into two parts:

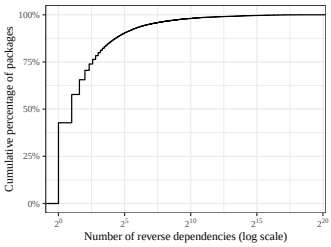
1. Ecosystem Metadata: This includes the full list of packages (2,663,681 packages), versions of every package (28,941,927 versions), and metadata for every version including version upload times, version numbers, dependencies, descriptions,



(a) An ECDF of the time in days between the publication of two versions of a package. Note that this plot specifically excludes updates for non-prerelease versions of packages.



(b) An ECDF of the number of (transitive) dependencies of each package. This was collected by resolving the latest version of every package on NPM as part of the experiment in Fig. 3.5.



(c) An ECDF of the number of reverse (transitive) dependencies of each package. Note that the x-axis is log-scaled. This was collected by resolving the latest version of every package on NPM as part of the experiment in Fig. 3.5.

Figure 3.2: ECDF plots of general properties of the NPM ecosystem with regards to versioning and dependencies.

links to repositories, and more. We also have a full scrape of all security advisories for NPM packages, including data on which versions are vulnerable and which version(s) patch the vulnerability.

2. Tarballs of published packages: The full source tarball of every version of every package[3] has been downloaded by our system.

Before diving into the core research questions, we first discuss general properties of the dataset. Fig. 3.2 displays three distributions regarding our main objects of interest: updates and dependencies. Fig. 3.2a displays an ECDF (empirical cumulative distribution function) of the distribution of the time between updates of packages, computed across 1,401,510 packages and 16,547,653 mined updates (Section 3.2.2). A surprising finding is how quickly updates are pushed out in many cases, with 25% of updates spanning only 39.87 *minutes* or less, and 50% of updates spanning 22.71 hours or less. However, a long tail of updates exists, with the top 25% of updates spanning 7.78 days or longer, and 10% spanning 40.12 days or longer. On average, updates span

3 excluding deleted content, which we describe in Section 3.6

21.03 days. A manual inspection of the data suggests that update behavior is quite bursty, with developers releasing multiple updates in rapid succession, and then going silent for long periods of time; however, the underlying cause of this behavior could be investigated more thoroughly.

Figs. 3.2b and 3.2c display ECDFs of the distributions of the numbers of (transitive) dependencies and downstream packages (i.e. transitive reverse dependencies), respectively. We selected the most recent non-prerelease version of every package with at least one update (to filter out abandoned packages), yielding 1,401,510 packages. We then used our time-traveling variant of NPM to resolve their dependencies and collect transitive dependency relations between packages, disregarding versions. Solving dependencies failed on some packages, due to both true solving failures with NPM (e.g. missing dependencies) and transient system failures (discussed more in Section 3.6) in the compute cluster. In total, our experiments include successful executions of NPM's dependency solver on 696,419 packages. The data shows that on average packages have 167.87 dependencies, and 95% of packages have solution sizes of 636 or fewer dependencies, with the largest solutions reaching up to 1,641 dependencies.

When turning to downstream packages however (Fig. 3.2c), the situation is quite asymmetrical, as there is a vastly longer tail of packages with massive amounts of downstream packages. The top 3 depended-upon packages that we observed were: a) `supports-color` (does a terminal support color?, 624,883 downstream packages), b) `debug` (logging library, 571,547 downstream packages), and c) `ms` (time conversion library, 515,684 downstream packages). On the other hand, a large amount of packages are unused except by a handful of downstream packages, with 50% of packages having 2 or fewer downstream packages, and 90% only being used by 30 or fewer downstream packages.

3.4.2 *RQ1: Version Constraint Usage*

As described in Section 3.2.1, developers can specify version constraints in different ways, which controls the installation of newer versions of those dependencies. Fig. 3.3 shows the frequency of each main type of version constraint published in each year since 2010, the year that NPM launched. For each year, we include only packages that had at least one release published, and if a package released multiple versions in that year, we include only the most recently published non-prerelease version that year. In

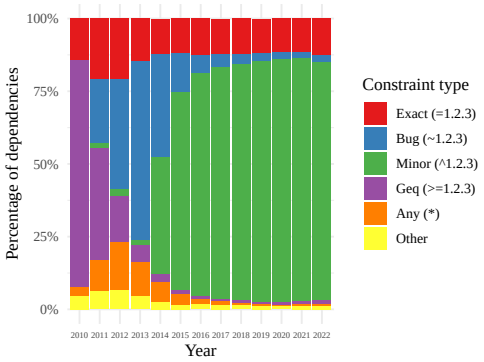


Figure 3.3: The relative popularity of each version constraint type across time. Percentages are *not* cumulative over past years, reflecting only published dependencies within each year.

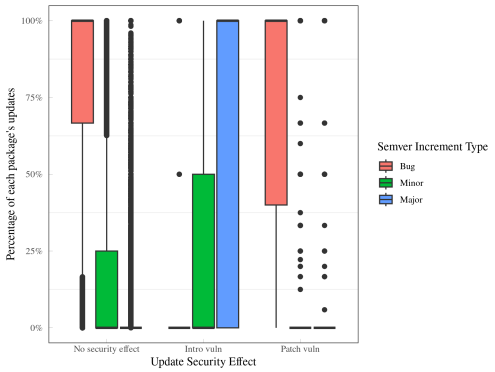


Figure 3.4: A boxplot visualizing the distribution of percentages of packages' updates by semver increment type, segmented across security effects. Within each security effect the percentages across semver increment types are normalized.

2022, there were a total 429,265 packages with at least one release, and across all the years 1,678,681 distinct packages.

There are several interesting trends in constraint usage over time. First, about 78.36% of all initial dependencies were specified as accepting any versions greater than some particular version (Geq, purple bars), such as `"react" : ">= 1.2.3"`. Developers then abandoned using Geq constraints within the first 3-4 years of NPM, likely because they became unmaintainable as libraries began to introduce breaking changes that would be automatically applied by Geq constraints. Second, even though constraints that are flexible in the minor component (Minor, green bars) currently represent a majority of dependencies, the phenomenon of using minor flexible constraints only started in 2014, and then rapidly expanded after. The expansion of minor flexible constraints coincides with the decreased usage of bug component flexible constraints (Bug, blue bars). Third, developers have recently gravitated towards using only two types of constraints almost exclusively: exact version constraints (Exact, red bars) and minor component flexible constraints. Together, those types represent over 94.85% of constraints in 2022. Finally, the percentage of dependencies that are potentially able to automatically receive updates

(everything below the red bars) has stayed relatively stable throughout the entire life of NPM, and is currently about 87.32% of all dependencies.

3.4.3 *RQ2: Semantic Versioning in Updates*

While RQ1 examined the usage of semantic versioning when specifying dependencies, RQ2 examines the usage of semantic versioning in deploying releases of those dependencies. Fig. 3.4 displays boxplots where each observation represents what percentage of a package's updates are one of the three semver increment types, normalized across security effect. This analysis includes 1,401,510 packages and 16,547,653 updates, as described in Section 3.2.2.

We find that in the no security effect category (the vast majority of updates), the most common updates by far are bug semver increments, with 75% of packages having 66% or more (lower quartile of left-most red box). Next most popular are minor semver increments, and finally least most popular are major semver increments.

However, when we consider updates that introduce vulnerabilities, we see a different story. Most packages introduce vulnerabilities via major semver increments, indicating that vulnerabilities are often introduced when packages developers release major new versions possibly consisting of many new features and significant structural changes to the code base. We did however find 29 outlier packages that introduced a vulnerability in at least one bug update. A particularly interesting example is an update to the `ssri` package (a cryptographic subresource integrity checking library, 23M weekly downloads) from version 5.2.1 to 5.2.2. The update attempted to patch a regular expression denial of service vulnerability, but inadvertently increased the severity of the vulnerability by changing the worst-case behavior from quadratic to exponential complexity [11]. This highlights the challenge package developers face in needing to quickly release patches to vulnerabilities, while needing to be extremely careful when working on security-relevant code and releasing it through bug updates that will be easily distributed to downstream packages.

Finally, in the case of vulnerabilities being patched, almost all patches are released as bug semver increments, which means that the 87.32% of non-exact constraints shown in Fig. 3.3 would potentially be able to receive them automatically. However, a handful of outlier packages have released vulnerability patches as non-bug updates (we found 358 such updates across 298 packages). From manual inspection, it appears that many of these updates include the fix for the security vulnerability mixed in with many other

changes, rather than the vulnerability fix being released independently. For example, update 1.6.0 to 1.7.0 of the `xmlhttprequest` package (1.2M weekly downloads) fixed a high-severity code injection vulnerability [21]. The security-relevant part of the update is only 1 line, but 892 lines were modified in the update. Without further investigation we do not know why some developers have chosen to include security patches as part of larger updates rather than as standalone updates.

3.4.4 *RQ3: Out-of-Date Dependencies and Update Flows*

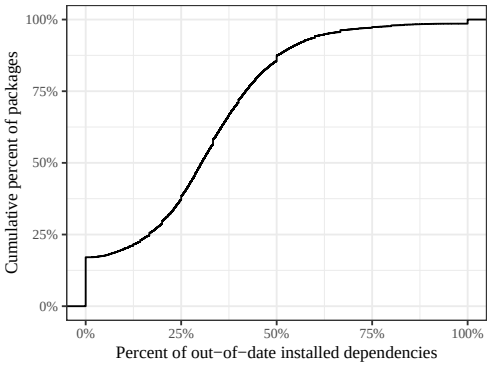
3.4.4.1 *How out-of-date are packages' dependencies?*

Version constraints and semver update types work in tandem to control the flow of updates to downstream packages, across many chains of transitive dependencies. Whether a downstream package receives up-to-date dependencies depends not only on the constraints at the downstream package and the type of semver increment at the reverse dependency, but also on packages in the middle of a transitive dependency chain.

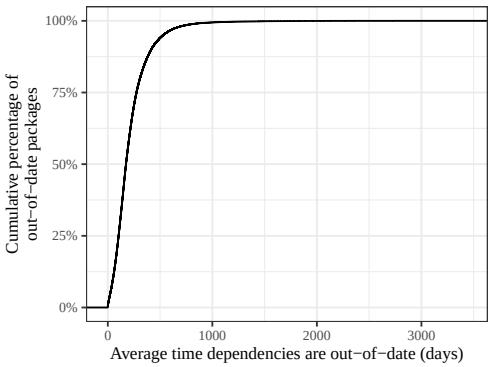
In this experiment, we select the latest version of every package with at least one update (1,401,510 packages). We then use our time-traveling variant of NPM to solve the package's dependencies at the time the latest version was uploaded (T_P). We then observe which of its installed dependencies are out-of-date, where a dependency with version V_D and upload time T_D is out-of-date if another version V'_D of the dependency has an upload time T'_D such that $T_D < T'_D < T_P$ and $V_D < V'_D$. We then define the out-of-date time as $T'_D - T_D$ for the largest such T'_D. After accounting for transient system failures, 696,419 packages were solved successfully.

Fig. 3.5a displays an ECDF of the distribution of the percentage of each package's dependencies that are out-of-date. There is a group of packages, about 17.08%, that have fully up-to-date dependencies. However, almost all of these have very few dependencies, only 3.17 dependencies on average compared to 167.87 dependencies for the whole sample. In other words, these fully up-to-date packages are packages that live primarily on the far left side of the ECDF in Fig. 3.2b.

Moving beyond the spike of up-to-date packages, most packages have at least some out-of-date dependencies, with 62.94% of packages having 25% or more of their dependencies out-of-date. Not only are packages often out-of-date, but they are often out-of-date for quite a while. Among packages with at least one out-of-date dependency, Fig. 3.5b displays an ECDF of on average how out-of-date each package's dependencies



(a) ECDF of percentage of each package's dependencies that are out-of-date.



(b) ECDF of average amount in days that dependencies are out-of-date by for each package with at least one-out-of-date dependency.

Figure 3.5: ECDF plots of technical lag distributions across the NPM ecosystem.

are. Half of all packages with out-of-date dependencies have on average dependencies that are 173.87 days old or older, with a long tail of 5% of packages with dependencies that are on average 527.38 days old or older. In contrast, updates are released within 21.03 days on average, and 50% are released within only 22.71 hours (Fig. 3.2a).

There can be a variety of reasons why packages have out-of-date dependencies, some of which are intentional, such as developers choosing to stay on older versions of libraries rather than rewrite code to handle breaking changes.

3.4.4.2 *How rapidly do updates flow downstream?*

We now wish to understand how updates flow to downstream packages, and how developers respond when manual intervention is required. For the most recent update prior to 2021 of every package, we randomly selected 50 downstream packages that were up-to-date with the upstream package just prior to the update. Using our time-traveling resolver we then solve the downstream package immediately after the update, and in 1 day increments afterwards until the dependency on the old version of the dependency has been updated or deleted. In total, 888,294 update flows were successfully solved after accounting for transient system failures.

Fig. 3.6 visualizes the process of how updates flow to downstream packages and how often developer intervention is required. An update flow has multiple steps. First,

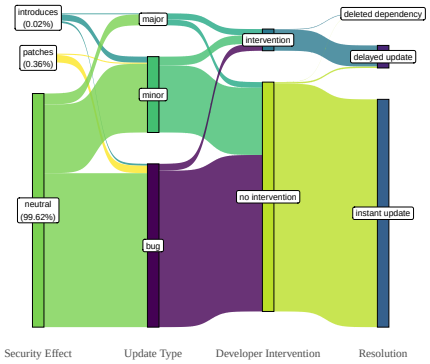


Figure 3.6: Visualization of update flow paths.

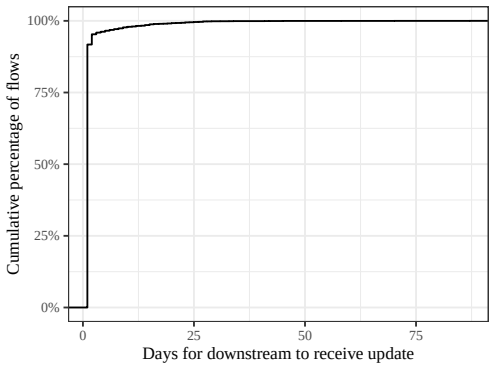


Figure 3.7: An ECDF plot of how long it takes for an update flow that is blocked to be resolved.

the upstream (dependency) developer publishes the update with a certain semver increment type (major, minor, or bug). Once the update is marked as bug, minor, or major and uploaded to NPM, it can then be received by downstream packages that depend on it, possibly transitively. This can happen either automatically, by the downstream developer manually updating or removing the dependency, or a developer in the middle of the transitive dependency chain updating or removing the dependency.

Most commonly, downstream packages receive the update instantly and with no human intervention needed ($\cdots \rightarrow$ no intervention $\rightarrow$ instant update). This occurs when the package that declares the constraint on the updated package uses a constraint that is at least as flexible as the type of semver increment. Note that the package declaring the constraint, and thus responsible for allowing or inhibiting the update flow, could be either the final downstream package or a package in the middle of the dependency chain. This type of flow occurs for the majority of bug and minor updates, which is induced by the distribution of constraint types (Fig. 3.3). As this type of flow is 90.09% of all analyzed update flows, it is by far the most common, indicating overall positive health among our random sample of update flows through the NPM ecosystem.

The second most common update flow consists of updates that require intervention from the developer of the downstream package (and possibly developers of other packages as well), and thus is delayed ($\cdots \rightarrow$ intervention $\rightarrow$ delayed update). This occurs in 9.01% of all analyzed update flows, and involves a major update 28.11% of

the time, a minor update 40.27% of the time, and a bug update 31.62% of the time. Updates requiring intervention are due to constraints that are more restrictive than the semver increment type. Intervention thus involves developers either switching to a more flexible constraint type or incrementing the constraint.

A small fraction (0.60%) of updates are resolved not by the developer of the downstream package performing an intervention, but by a developer(s) in the middle ($\cdots \rightarrow$ no intervention $\rightarrow$ delayed update). For this to occur, the developer of the package in the middle must have specified a constraint that is too restrictive, while the developer of the downstream package specified a flexible enough constraint to allow for the intervention of the package in the middle to be adopted. Since this type of flow happens very rarely, this indicates that downstream packages typically use constraint tyes that are equally or more restrictive than the types of constraints their (transitive) dependencies use. This makes sense from a software engineering perspective, as the deeper packages (those closer to libraries rather than applications) have more incentive to use flexible constraints as they are likely to be reused in contexts with otherwise conflicting constraints.

The final type of flow is when the out-of-date dependency is eventually deleted rather than updated ($\cdots \rightarrow$ intervention $\rightarrow$ deleted dependency). This occurs in only 0.29% of all analyzed update flows, indicating that developers do not generally delete dependencies.

Among the update flows that are blocked due to restrictive constraints, almost all update flows are unblocked via manual intervention quite rapidly. Fig. 3.7 shows an ECDF of the distribution of how many days it takes for each update flow to be unblocked. The majority of blocked update flows (91.74%) are unblocked within 1 day, with a tail trailing off to 25 days or more. The surprising speed of update flows being unblocked is due largely to the fact that many packages that depend on each other are developed by the same contributors, and they will often bump version numbers and update dependencies of their packages nearly simultaneously.

Our results suggest that most updates effectively flow to downstream packages, while Fig. 3.5 suggests that most downstream packages have at least some out-of-date dependencies. We suspect this is due in part to the number of dependencies per-package (Fig. 3.2b) and the rate of updates (Fig. 3.2a). With packages having an average of 167 dependencies, and updates being released on average every 21 days, we would expect that for an average package every day multiple dependencies release updates and potentially go out-of-date. Even with many updates being adopted instantly or quickly, some dependencies will become stale. This phenomena might also be explained by

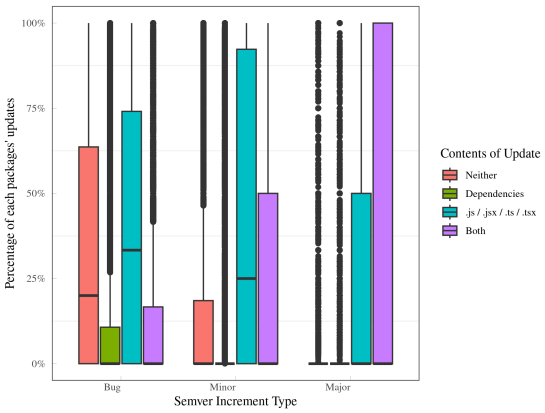


Figure 3.8: A boxplot displaying the distribution of the percentage of packages' updates grouped by semver increment type that change only code (`.js`, `.ts`, `.jsx`, `.tsx`), only dependencies, both, or neither.

our methodology for this experiment, as we selected only packages that were already up-to-date at the time of our analysis.

3.4.5 *RQ4: Analyzing Code Changes in Updates*

We now turn to inspecting the *contents* of package updates rather than metadata analysis. Semantic versioning can only be useful if package developers release updates that are in accordance with what downstream packages expect from bug, minor, or major semver increments. In this work, we focus on providing a high-level characterization of what updates generally consist of in the NPM ecosystem, across the different update types. While more fine-grained analyses and related applications would be interesting and useful, it is beyond the scope of this work, and we defer discussion of ongoing and future work to Section 3.5. However, we believe that our dataset may be a useful building block for evaluation within the active research area of update analysis systems.

Fig. 3.8 displays a boxplot where each observation is the percentage of a package's updates within each semver increment type that change only code (`.js`, `.ts`, `.jsx`, `.tsx`), only dependencies, both, or neither. Note that updates categorized as neither may include other changes such as modifications to other file types (README, CSS, etc.) or other metadata changes besides dependencies. This uses the same set of packages

and updates as from Fig. 3.4, intersected with those we were able to successfully download tarballs for, giving in total 1,339,684 packages and 14,903,021 updates.

First, we see that bug updates often contain no changes to code files, or to dependencies. 50% of packages change neither code nor dependencies in about 20% or more of their bug updates, while 25% of packages change neither in a majority (64%) of their bug updates. A manual inspection of the data suggests that some of these updates consist of changes to metadata (listed contributors, descriptions, READMEs) or to configuration files (`.json`, `.yaml`, etc.), while other updates truly change nothing. However, more investigation on our data could be done to quantify this more precisely. Second, while it is not common to do so, 25% of packages do occasionally release bug updates which only modify dependencies (11% or more of bug updates). Looking at minor and major updates, the frequency of packages modifying neither or only one or the other decreases, and when looking at major updates, most packages modify both code and dependencies simultaneously. In combination with the results of RQ2, we see that overall, minor and major updates appear to present a higher degree of risk (larger changes and/or security vulnerabilities) than bug updates.

3.5 DISCUSSION

Considering the results that we presented in Section 3.4, we find a number of implications for software developers, ecosystem maintainers and researchers. Developers consuming dependencies face persistent trade-offs between security, reliability, and technical lag. We identify opportunities for ecosystem maintainers to reduce some of this friction and point towards longer-term research directions to address some of the underlying challenges in package ecosystems.

3.5.1 *For Developers*

Our findings for RQ1 indicate that NPM has largely consolidated around using either exact or minor-flexible (^) constraints, with the greatest proportion of dependencies specified as minor-flexible. In practice this means that minor updates will flow to downstream packages nearly as easily as bug updates, which we confirmed experimentally in RQ3.2, with 95.42% of sampled bug updates and 86.55% of sampled minor updates flowing automatically to downstream packages. This finding is important for library

maintainers, who might expect that downstream packages will manually inspect minor updates for compatibility.

Overall, there is a misalignment between the way that versions are released and the way that they are depended on, as versions that are released as minor vs. bug updates commonly have distinct characteristics (Figs. 3.4 and 3.8), while dependencies in downstream packages rarely distinguish between minor or bug updates (Fig. 3.3). Specifically, we find that 81.19% of updates are released as bug updates, but 84.01% of dependency constraints accept bug and minor updates. While both bug and minor updates are supposed to maintain backwards compatibility, since minor updates may be more likely to include (inadvertent) breaking changes, developers may benefit in stability by using bug-flexible (~) constraints rather than minor-flexible constraints, which would still receive 81.19% of updates. This motivation may be even stronger for security-cautious developers as our results suggest that minor updates introduce vulnerabilities more often than bug updates, however they must remain careful as even bug updates occasionally introduce vulnerabilities.

3.5.2 *For Ecosystem Maintainers*

Our findings in RQ2 indicated that some developers release security patches with minor and sometimes, even major version increments. This finding is concerning as it makes it more difficult for downstream packages to receive the security fixes. This suggests that ecosystems may benefit from ecosystem maintainers attempting to have tighter communication with package developers around security patches, and help ensure that security patches are released in a timely manner, with minimal changes, and as semver bug updates.

Our findings in RQ3.2 show a small fraction of update flows that are blocked by dependencies in the middle. For example, the `vt-pbf` package (a geographic data file serializer, 900K weekly downloads) has a exact version constraint on the `@mapbox/point-geometry` package, and thus consumers of the `vt-pbf` package will not receive updates to `@mapbox/point-geometry` unless `vt-pbf` specifically releases an update bumping the version number. This is perhaps the most frustrating case for developers, as it is difficult to remedy the situation. One option is to use NPM's overrides feature [74], which allows the downstream package to forcefully override versions of transitive dependencies, even if this breaks version constraints. While this can be effective in the short-term, one challenge is that the developer now has the maintenance

burden of removing the override when it is no longer necessary, or else face out-of-date dependencies in the future. To improve the developer experience, ecosystem maintainers could a) reduce the frequency of update propagation blockage by combining our analysis with centrality analysis to find critical packages that often block update flows, and work with them to address the situation; and b) improve ecosystem tooling around overrides to help developers automate the removal of overrides when no longer necessary.

3.5.3 *For Researchers*

Our findings in RQ2 indicate that while NPM developers generally try to follow semver conventions, they do not always do so consistently, and thus developers of downstream packages can not be entirely confident about what exactly they will receive when updating dependencies (particularly if malicious developers release malware!). This suggests a useful and broad design space of static or dynamic program analysis tooling that could help give insight on what actually changes in an update. Such tools could aim to check for semver compliance [54, 77], check that an update actually patches a vulnerability correctly, check for likely buggy changes in behavior [107], or detect malware. It may be particularly interesting to examine trends in semver compliance over time, as our analysis shows clear trends in the changing popularity of dependency constraints between 2010–2022.

There is already promising ongoing work in some of these directions, particularly malware detection [94, 95, 111] via metadata and lightweight syntactic features. In RQ4 we found that a significant portion of packages publish bug updates that change neither `js`, `ts`, `jsx`, `tsx` files, nor dependencies, which suggests that a sizeable portion of updates may be changing other types of files, and that these changes may be an effective place for bad actors to hide malicious changes, such as introducing malicious changes to shell scripts embedded within `.yaml` continuous integration configuration files. Whether the aim of this work is malware detection, bug detection, or other analyses, our results suggest that such tooling should aim to handle multiple file types, such as code, config files, embedded binaries, shell scripts, etc.

The analysis in RQ3.1 finds that there is a substantial amount of technical lag in NPM packages, so tooling to help developers reduce technical lag could be quite impactful. In Chapters 4 and 5 we build a tool, MaxNPM, which allows developers to solve dependencies in a way that minimizes technical lag (or other objectives) while still

satisfying current version constraints, but does not help when constraints themselves are out-of-date. Complementary future research, such as what Jayasuriya suggests [50], could assist developers in performing these manual updates by helping with code migration in response to breaking changes.

3.6 THREATS TO VALIDITY

3.6.1 *External Validity*

We were unable to reliably scrape packages that have been deleted (for malware, copyright violations, etc.) or unpublished (voluntarily by the developer) from NPM, and thus we excluded these in our analyses. For this reason our results might not generalize to malware or other types of packages that are often deleted.

Other than deleted packages, we consider the entire ecosystem, including so called "trivial" packages [16, 52] and packages that seem unimportant (e.g. few reverse dependencies). We believe that it is difficult to tell if a package truly is irrelevant, as even a package with very few reverse dependencies may in fact be an application that has been published on NPM. Furthermore, "low-impact" developers are nevertheless important as their experience with NPM matters for the future of the ecosystem.

We only obtain packages from NPM, and do not consider GitHub or other sources. As such, this study may not generalize to JavaScript applications (rather than libraries), as only some developers choose to publish their applications on NPM. In addition, some developers may include dependencies by directly copying source files into their packages, which we do not detect. Finally, it is important to be careful when generalizing our results about security vulnerabilities, as we are only able to obtain information about *known* vulnerabilities, which is likely a small subset of all vulnerabilities.

3.6.2 *Internal Validity*

Our system described in Section 3.3 is complex and it is possible that there are bugs in our system that could affect the results of our experiments. For example, we may have missed some packages in our scraping process, or we may have incorrectly downloaded some packages. We believe that this is unlikely, as we have written unit tests for our system and have tested it on a small subset of packages, and have not found any bugs.

While running millions of package installations for RQ3 (Section 3.4.4) we caused intermittent failures on our compute cluster by overflowing `/tmp`. Since these failures were a function of system state and not of packages, we do not believe this biased our results. To check this, we computed the mean and median of the number of direct dependencies of successful and failed packages, and found that successful packages have a mean of 9.91 direct dependencies and a median of 5, while the failed packages have a mean 10.93 direct dependencies and a median of 6. This suggests that failed packages were a bit larger, but not enough to make our successful packages unrepresentative. Outliers with a large number of dependencies existed with both failed and successful packages.

3.6.3 *Construct Validity*

Throughout RQ2–RQ4 we use our algorithm for computing updates as described in Section 3.2.2. Since there is no ground truth for correctly mined updates, one may wish to consider refinements to this algorithm. In particular, one may wish to have fine-grained equivalence classes by considering minor components as well. However, this would not change the results where our algorithm already succeeds, and since the rejection rate is already quite low (1.8%) we did not believe a more complex algorithm justified the risk of analysis bugs.

In RQ4 we defined code changes to mean files with extensions `.js`, `.ts`, `.jsx`, or `.tsx`. This is because we wanted to focus on JavaScript and TypeScript code, but this may have caused us to miss some JavaScript or TypeScript code with other extensions. Depending on the purpose, future work might want to consider a broader definition of what counts as code, such as shell scripts.

3.7 CONCLUSION

We present a large-scale analysis of semantic versioning in NPM, and a full, reusable dataset of complete package metadata and tarball data from NPM. We find that there is a higher risk of security vulnerabilities being introduced through minor rather than bug (i.e. patch) semver updates, suggesting a motivation for developers to use bug-flexible constraints (`~`), even while the NPM ecosystem has largely abandoned them in favor of minor-flexible constraints (`^`). While we find that most security patches are introduced in bug updates, we find a disturbing set of outliers that are released as minor or even

major updates, potentially causing slower adoption of security patches. Future work examining the NPM ecosystem might build on our dataset and tooling, examining the contents of updates for bugs and/or vulnerabilities, along with mechanisms to mitigate technical lag.

3.8 DATA AVAILABILITY

Our artifact permanently archived on Zenodo [82] contains our tools and the metadata from our dataset.

4

PACSOLVE: A FORMAL MODEL OF DEPENDENCY MANAGEMENT

4.1 THE LANDSCAPE OF DEPENDENCY SOLVING

The design space of existing dependency solvers is large and varied. To better understand the subject we wish to formally model, we first describe several features that dependency solvers share, and highlight differences in their semantics.

4.1.1 *Versions and Constraints*

In modern package managers, programmers typically write constraints (often ranges) that describe allowable versions of dependencies. These constraints allow the dependency solver to semi-automatically update dependencies and to unify dependencies to share code. For example, suppose `foo` and `baz` both require `bar` and constrain the version to be $\texttt{1}.n.m, \forall n, m \in \mathbb{N}$. This kind of constraint would allow the dependency solver to find a single version of `bar` for both packages. This is a common type of constraint: version `2.0.0` may break compatibility with versions $\texttt{1}.n.m$.

Unfortunately, it is not always straightforward to express this constraint to a dependency solver. Each dependency solver has its own little DSL for specifying version constraints. However, these DSLs can behave in surprising ways. In the NPM ecosystem, we can write $\texttt{1}.n.m$ as follows (spaces indicate conjunction):

```
"bar": ">=1.0.0 <2.0.0"
```

This constraint works as expected. We can also write a range constraint in Maven, as a half-open interval:

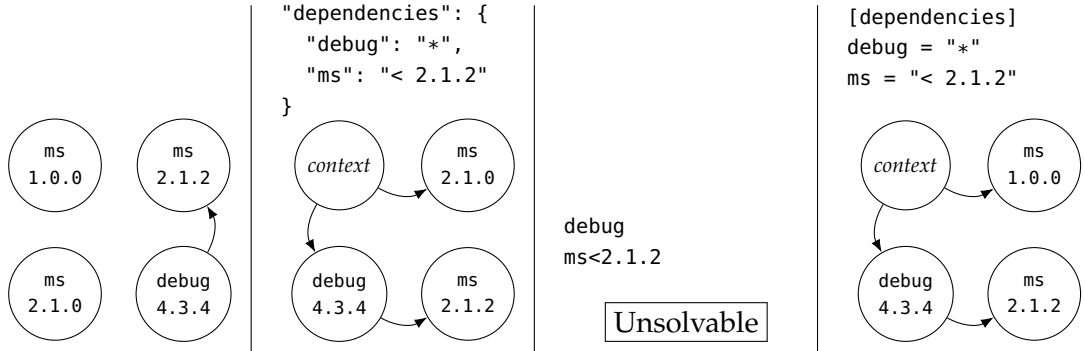
```
[1.0.0, 2.0.0)
```

Unfortunately, this constraint does not work as expected. Suppose there is a pre-release version of `bar` numbered `2.0.0-alpha-1` in the package repository. If so, *Maven will install the incompatible pre-release package*, because $\texttt{2.0.0-alpha-1} < \texttt{2.0.0}$. This

interpretation of constraints makes it difficult to use ranges in Maven [53]. In contrast, NPM interprets ranges differently and would not install the pre-release version [72].

4.1.2 *Version Conflicts*

What should a dependency solver do when conflicting constraints occur? Different dependency solvers address conflicts in very different ways. To illustrate, we consider a tiny subset of the `npmjs.com` repository (Fig. 4.1a): three versions of the package `ms` (a collection of calendar functions), and one version of the package `debug`, which depends on the latest version of `ms`. Suppose a programmer requires both `debug` and `ms`, but notices a performance regression in the latest version of `ms`. They may reason that their code should use an older version of `ms`, but it is okay for their debugging code, which is not performance-sensitive, to continue to transitively require the latest version that has the performance regression.



(a) Available packages (b) JavaScript/NPM (c) Python/PIP (d) Rust/Cargo

Figure 4.1: The `debug` package (a debug logging library) and `ms` package (a library to convert times values to miliseconds) are available on `npmjs.com`, with 16.4 billion and 12.6 billion total downloads, respectively. Fig. 4.1a shows a subset of the versions of each package, and `debug`'s exact version constraint on `ms`. Figs. 4.1b to 4.1d illustrate the three different results generated by NPM, PIP, and Cargo if the programmer were to ask for any version of `debug` and any version of `ms` strictly less than `2.1.2`, supposing the versions shown in Fig. 4.1a are all the versions in the universe.

Given these constraints, NPM installs both versions `2.1.0` and `2.1.2` of `ms` (Fig. 4.1b). In general, NPM allows several versions of a package to be co-installed and loaded at

runtime.[1] However, this behavior is not always desirable, and can lead to increased code size, and subtle runtime bugs. Moreover, NPM does not guarantee that packages are only duplicated when strictly necessary.

Suppose NPM used PIP's dependency solver that does not allow duplicate packages. Moreover, PIP backtracks to find a solution if necessary. In this example, PIP would report that the dependencies are unsolvable (Fig. 4.1c). While this is an unfortunate outcome for the programmer, PIP behaves this way because the Python module system cannot load multiple package versions of the same package. However, an advantage of this approach is that it reduces code size and avoids compatibility problems.

Cargo attempts to strike a balance between NPM and PIP, by disallowing two *minor revisions* of the same package to be co-installed. In this case, versions `2.1.0` and `2.1.2` can't be co-installed, so Cargo would choose to co-install `1.0.0` and `2.1.2` (Fig. 4.1d). Arguably, this solution is worse than the NPM solution, since the program is forced to directly depend on a much older version of `ms`. Like PIP, Cargo backtracks to find a solution, if one exists.

Unfortunately, it is not clear which (if any) of these policies are the best, and programmers in different scenarios may want different policies. A goal of PacSolve is to make these choices more transparent via a formal semantics, and ultimately configurable by the programmer.

4.1.3 *Optimization Objectives*

Apart from satisfying dependency constraints, most dependency solvers also try to bias which versions of dependencies they choose. For example, PIP, NPM and Cargo all prefer to select newer versions rather than older versions of a dependency. These solvers all do so greedily, so the final solution depends on the order in which the dependency solvers explore the solution space. NPM and PIP are sensitive to the order in which developers list dependencies, whereas Cargo's dependency solver is sensitive to the lexicographic order of package names. Regardless, all of these dependency solvers optimize greedily, and do not attempt to find globally optimal solutions.

A more fundamental problem is that what it means to be a *newer package* is not well defined. Suppose package A has versions `1.0.0` and `2.0.0`, and then its author publishes a security update to the older version numbered `1.0.1`. What happens if a program depends on A with no constraints? Cargo and PIP will both prefer to choose

1 The exception to this rule are *peer dependencies*, which are not automatically installed.

`2.0.0` since it has a larger version number, whereas NPM will choose `1.0.1` because it was uploaded last.

4.1.4 *Solution Spaces*

Dependency solvers use different data structures to represent the solution space. Linux system package managers (e.g. APT), typically model installed packages as a *set*, e.g. `libpng` either is or isn't installed. PIP behaves in a similar way, and requires virtual environments to install multiple versions of a package. Prior work on using SAT solvers for package management encoded a solution space of installed package sets into SAT formulas, with one boolean variable per installable dependency [101].

However, a set representation is inadequate for NPM and Cargo. Consider again the solution presented in Fig. 4.1b. The dependency solver has to keep track of not only that `ms` versions `2.1.2` and `2.1.0` are installed, but also ensure that each dependency on `ms` refers to the right version. This naturally leads to modeling solutions as directed graphs, which generalizes the set model.

Unfortunately, a directed graph is still not expressive enough for Cargo, which allows a package to explicitly and directly load several versions of the same dependency by giving each version a alias name [12]. To support this behavior, the dependency solver must be able to match each resolved dependency version with each constraint, and also handle the case where the two constraints resolve to the same version. We can represent these solutions as directed multi-graphs, generalizing the directed graph. Finally, an additional variable in solution space design is that Cargo disallows cycles in the solution graph, whereas NPM freely allows cycles.

4.1.5 *Why a Semantics?*

Package managers evolve, and existing package managers have made significant changes to their dependency solvers over time. It is easy to imagine building a new version of NPM that tries to find globally optimal solutions, or unify dependencies in the style of PIP. But, we believe that this would just be yet another point in the dependency solver design space, and not a principled approach to the problem.

Instead, we next present a parameterized semantics for dependency solving that makes it possible to compactly express a variety of different dependency solving

policies. Moreover, the semantics is executable, and we build on it in Chapter 5 and Chapter 6.

4.2 A SEMANTICS OF DEPENDENCY SOLVERS

This section first presents PacSolve. We then give some examples of dependency solver specifications for PacSolve.

4.2.1 *A Relational Semantics of Dependency Solvers*

In the abstract, a dependency solver can be thought of as a function that receives as input a) metadata about available packages from package repositories, and b) a root package to install. Its output is a *solution graph* whose nodes are particular package versions drawn from the set of available packages. Hidden in this function are design decisions summarized in Section 4.1. In contrast, we formulate PacSolve as a relation (Fig. 4.2) to account for dependency solvers that may admit multiple solutions. $\mathcal{S}$ is a ternary relation between package metadata ($\mathcal{M}$), a dependency solver specification ($\mathcal{F}$), and a solution graph ($\mathcal{G}$).

PACKAGE METADATA The package metadata (top of Fig. 4.2) is a tuple that has 1) a set of package name and version pairs ($\mathcal{N}$), and 2) a finite map (*deps*) from these name-version pairs to a list of dependencies ($\mathcal{D}$). Each dependency specifies a package name and a version constraint ($\mathscr{C}$) on that package. The syntax of package versions ($\mathcal{V}$) and version constraints ($\mathscr{C}$) varies between dependency solvers, and their semantics is determined by the dependency solver specification, which we describe below. Without loss of generality, we assume that all package names are strings and that there is a distinguished root package (**root** $\in \mathcal{N}$).

DEPENDENCY SOLVER SPECIFICATION The dependency solver specification is a 4-tuple with the following components:

1. A *constraint satisfaction predicate* that consumes a constraint and a package version, and determines if that version satisfies the constraint (*sat*);
2. A *version consistency predicate* that consumes two package versions and determines if those two versions of the same package may be co-installed (*consistent*);

Package Metadata

$\mathscr{P} ::=$	**String**	Package Names
$\mathcal{V}$	is a set	Version Numbers
$\mathscr{C}$	is a set	Version Constraints
$\mathcal{D} ::=$	$\mathscr{P} \times \mathscr{C}$	Dependencies
$\mathcal{N} \subseteq$	$\mathscr{P} \times \mathcal{V}$	Package repository (finite set)
$deps :$	$\mathcal{N} \cup \{\textbf{root}\} \xrightarrow{\text{fin}} \mathcal{D}^*$	Dependencies per node
$\mathcal{M} ::=$	$\langle \mathcal{N}, deps \rangle$	Package metadata

Dependency Solver Specification

$sat :$	$\mathscr{C} \to \mathcal{V} \to$ **Bool**	Constraint satisfaction semantics
$consistent :$	$\mathcal{V} \to \mathcal{V} \to$ **Bool**	Version consistency versions
$cycles\_ok \in$	**Bool**	If cycles are permitted in solution graphs
$minGoal :$	$\mathcal{G} \to \mathbb{R}^n$	Objective functions
$\mathcal{F} ::=$	$\langle sat, consistent, cycles\_ok, minGoal \rangle$	Dependency solver specification

Solution Graph

$N_R \subseteq$	$\mathcal{N} \cup \{\textbf{root}\}$	Package versions in solution
$D_R \in$	$N_R \to N_R^*$	Solved dependencies
$\mathcal{G} ::=$	$\langle N_R, D_R \rangle$	Solution graphs

The PacSolve Relation

$\mathcal{S} \subseteq 2^{\mathcal{F}} \times 2^{\mathcal{M}} \times 2^{\mathcal{G}}$

For all $(\langle sat, consistent, cycles\_ok, minGoal \rangle, \langle \mathcal{N}, deps \rangle, \langle N_R, D_R \rangle) \in \mathcal{S}$:

1. $\textbf{root} \in N_R$
2. $\langle N_R, D_R \rangle$ is connected
3. $\forall n.n \in N_R \implies |D_R(n)| = |deps(n)|$
4. $\forall n.n \in N_R \implies \forall i.0 \leq i < |D_R(n)| \implies \exists p, v, c.\ (p, v) = D_R(n)[i] \wedge (p, c) = deps(n)[i] \wedge sat(c, v)$
5. $\forall p, v, v'.\ (p, v), (p, v') \in N_R \implies consistent(v, v')$
6. $\neg cycles\_ok \implies \langle N_R, D_R \rangle$ is acyclic

Figure 4.2: The PacSolve Model of Dependency Solving

3. A flag that determines if the solution graph may have cycles (*cycles_ok*); and
4. An *objective function* that consumes a solution graph and produces its cost (*minGoal*).

SOLUTION GRAPHS The result of a dependency solver is a solution graph ($\mathcal{G}$). A solution graph is a directed graph, where the nodes are package-version pairs, and each node has an ordered list of edges. The order of edges corresponds to the order of dependencies in the package metadata. For example, suppose the dependencies of **root** are the packages A, B, C (with some constraints) in the package metadata. If so, the outgoing edges from **root** in the solution graph would refer to specific versions of A, B, C in the solution graph.

SEMANTICS The semantics of PACSOLVE is a relation ($\mathcal{S}$) that holds when a solution graph is valid with respect to the package metadata and the dependency solver specification. The relation holds if and only if the following six conditions are satisfied. First, the solution graph must include the root:

$$\textbf{root} \in N_R$$

Second, the solution graph must be connected, to ensure it does not have extraneous packages:

$$\langle N_R, D_R \rangle \text{ is connected}$$

Third, for all packages in the solution graph, every edge must correspond to a constraint in the package metadata:

$$\forall n.n \in N_R \implies |D_R(n)| = |deps(n)|$$

Fourth, for every edge in the solution graph that points to package p with version v, the corresponding constraint in the package metadata must refer to package p with constraint c, where v satisfies c:

$$\forall n.n \in N_R \implies \forall i.0 \leq i < |D_R(n)| \implies \\ \exists p, v, c.\ (p, v) = D_R(n)[i] \wedge (p, c) = deps(n)[i] \wedge sat(c, v)$$

The criteria so far are adequate for many dependency solvers, but permits solutions that may be unacceptable. For example, without further constraints, a solution graph

may have several versions of the same package. Thus the fifth condition ensures that if there are versions of a package in the solution graph, then the two versions are consistent, as judged by the dependency solver specification:

$$\forall p, v, v'.\ (p, v), (p, v') \in N_R \implies \mathit{consistent}(v, v')$$

NPM allows arbitrary versions to be co-installed (so *consistent* is the constant true function), Cargo only allows semver-incompatible versions to be co-installed, and PIP only allows exactly one version of a package to be installed at a time (so *consistent* requires $v = v'$).

A final distinction between dependency solvers is whether or not they allow *cyclic dependencies*. NPM and PIP support them, whereas Cargo does not. Thus the sixth condition uses the dependency solver specification to determine whether or not cycles are permitted:

$$\neg \mathit{cycles\_ok} \implies \langle N_R, D_R \rangle \text{ is acyclic}$$

These six conditions determine whether or not a solution graph is correct with respect to the semantics of a particular dependency solver.

4.2.2 *Example: A Fragment of NPM in* PacSolve

We now present a dependency solver specification ($\mathcal{F}$) for a fragment of NPM. MaxNPM is a complete implementation that we discuss in Chapter 5.

In the concrete syntax of NPM, versions are represented as JSON strings. For example, the string `"x.y.z"` represents the version $x.y.z$ using *semantic versioning* (semver): each decimal represents a major, minor, and bugfix version, respectively. Incrementing a major version indicates a break in backward compatibility; incrementing a minor version indicates that features have been added, but none break existing APIs; and patch version differences indicate bugfixes that have no effect on the compatibility of two versions. NPM also uses strings to represent constraints, and supports a variety of operators, including conjunction, disjunction, upper and lower bounds, and semver compatibility. It is straightforward to translate these version and constraint strings to equivalent S-expressions: we represent a version as a list of three numbers, `(x y z)` and constraints as shown in Fig. 4.3a. With both constraints and versions written as S-expressions, we can write the version-constraint satisfaction predicate as a recursive function in Racket, which uses pattern matching to compactly interpret ranges and semver compatibility (Fig. 4.3b).

$\mathcal{V} \coloneqq (x\ y\ z)$	Version numbers
$\mathscr{C} \coloneqq (\texttt{=}\ x\ y\ z)$	Exact
$\mid *$	Any
$\mid (\texttt{<=}\ x\ y\ z)$	At most
$\mid (\texttt{>=}\ x\ y\ z)$	At least
$\mid (\wedge\ x\ y\ z)$	Semver compatible with
$\mid (\texttt{and}\ \mathscr{C}_1\ \mathscr{C}_2)$	Conjunction
$\mid (\texttt{or}\ \mathscr{C}_1\ \mathscr{C}_2)$	Disjunction

(a) Example of $\mathcal{V}$ and $\mathscr{C}$, allowing conjunction, disjunction, and range operators on semver-style versions.

```
(define (sat c v)
  (match `(,v ,c)
    [`((,x ,y ,z) (= ,x ,y ,z))        #true]
    [`(,_ *)                           #true]
    [`((,x ,y ,z1) (<= ,x ,y ,z2))     (<= z1 z2)]
    [`((,x ,y1 ,z1) (<= ,x ,y2 ,z2))   (< y1 y2)]
    [`((,x1 ,y1 ,z1) (<= ,x2 ,y2 ,z2)) (< x1 x2)]
    [`((0 0 ,z1) (^ 0 0 ,z2))          (= z1 z2)]
    [`((0 ,y ,z1) (^ 0 ,y ,z2))        (>= z1 z2)]
    [`((0 ,y1 ,z1) (^ 0 ,y2 ,z2))      #false]
    [`((,x ,y ,z1) (^ ,x ,y ,z2))      (>= z1 z2)]
    [`((,x ,y1 ,z1) (^ ,x ,y2 ,z2))    (> y1 y2)]
    [`(,_ (and ,c1 ,c2))               (and (sat c1 v) (sat c2 v))]
    [`(,_ (or ,c1 ,c2))                (or (sat c1 v) (sat c2 v))]
    [`((,x1 ,y1, z1) (>= ,x2 ,y2 ,z2)) (sat `(,x2 ,y2 ,z2) `(<= ,x1 ,y1 ,z1))]
    [_                                 #false]))
```

(b) A *sat* interpretation function for $\mathcal{V}$ and $\mathscr{C}$ defined in Fig. 4.3a.

Figure 4.3: Example of $\mathcal{V}$, $\mathscr{C}$, and *sat*

```
(define (minGoal-num-deps g)
  (length (graph-nodes g)))
```

(a) Minimize the total number of installed dependencies.

```
(define (minGoal-duplicates g)
  ; we count how many times
  ; each package name occurs
  (define package-counts (foldl
    (lambda (n counts)
      (define p (node-package n))
      (hash-set counts p
        (add1 (hash-ref counts p 0))))
    (make-immutable-hash)
    (graph-nodes g)))

  ; then assign a cost of 1
  ; for each duplicate
  (apply +
    (map
      (lambda (c) (max 0 (sub1 c)))
      (hash-values package-counts))))
```

(b) Minimize the total number of co-installed versions of the same package.

```
(define (minGoal-oldness g)
  (apply +
    (map
      (lambda (n)
        (get-oldness
          (node-package n)
          (node-version n)))
      (graph-nodes g))))

(define (get-oldness p v)
  ; The get-sorted-versions retrieves
  ; a list of all versions of p
  (define all-vs
    (get-sorted-versions p))
  (if (= (length all-vs) 1)
      0
      (/ (index-of all-vs v)
          (sub1 (length all-vs)))))
```

(c) Minimize the amount of "oldness" present in the solution graph. Each resolved dependency contributes an oldness proportional to its rank among the total ordering of versions of that package

Figure 4.4: Three different examples of PacSolve minimization objectives

4.2.3 *Example: Objective Functions*

The goal of a dependency solver is to find a solution that is not only correct, but also good by some metric. Some dependency solvers prefer to install versions with larger version numbers, whereas others prefer to install more recently uploaded versions. Other metrics are possible as well, such as total download size or number of dependencies [101] and multiple prioritized objectives [33, 35]. PACSOLVE takes an objective function that maps a solution graph to a sequence of numbers that represent a prioritized list of minimization criteria ($minGoal : \mathcal{G} \to \mathbb{R}^n$).

Figure 4.4, defines three examples of minimization functions, to give an example of concrete criteria. The first example (Fig. 4.4a) is the simplest: given a solution graph G, it returns how many nodes are in G, thus minimizing the total number of dependencies. The second example (Fig. 4.4b) minimizes the co-installation of multiple versions of the same package. The function first counts how many nodes there are in the graph for each package. Then, for each package, we assign a cost of 1 for each *extra* node, and sum all the costs up. The third example (Fig. 4.4c) is an interpretation of the common goal that package managers have of trying to choose newer versions of dependencies. In this example, each node in the graph is assigned an *oldness*, which is a linear score between 0 (newest) and 1 (oldest) of that version's rank in the total ordering of versions for that package. These oldness scores are then summed up across the graph. There are two subtleties with this definition. 1) The choice to perform *minimization* rather than maximization is important, as a goal of maximizing a newness score, e.g. 1 meaning newest, 0 meaning oldest, would encourage the solver to find very large solution graphs so as to inflate the total newness. 2) The choice to sum rather than average the oldness values likewise discourages the solver from adding in a lot of new nodes so as to inflate the mean oldness.

4.3 REASONING ABOUT DEPENDENCY SOLVERS WITH PACSOLVE

PACSOLVE gives us a way to write down the (missing) formal specification of a dependency solver. In this section, we sketch what a PACSOLVE-based semantics would look like for PIP, NPM, and Cargo. We then state some simple, universal properties that we believe any dependency solvers *should satisfy*. Notably, these are universal properties that are independent of an implementation. We then ask if the implementations of PIP, NPM, and Cargo satisfy these properties. Our testing suggests that PIP satisfies two

of three properties, while PIP and NPM only satisfy one property. Finally, we state additional properties about how we believe a package manager's dependency solver should relate to the semantics of the module system of its programming language. We summarize which of these properties are satisfied by popular language package managers in Table 4.1.

4.3.1 *Specifying Versions and Version Constraints*

A package manager can choose a syntax for version numbers that stores any data necessary to disambiguate configurations of a package. For example, an accurate structure of version numbers for NPM contains not only three numbers for a semver version, but an additional field for optional release tags: e.g., `1.2.3-alpha1`. Python's PIP package manager uses version numbers that are fully compatible with semantic versioning, with similar extensions for pre- and post-release versions [17].

Version numbers in Rust's Cargo are more sophisticated. Cargo allows packages to contain named *features*, which can be used to enable conditional compilation and enable otherwise disabled dependencies. For example, an image processing package may contain the features `"png"` and `"jpeg"`. To model this, we might choose elements of $\mathcal{V}$ to have the form (x, y, z, F), where F is a subset of the listed features for version $x.y.z$.

Similarly, constraints ($\mathscr{C}$) may be enriched to contain necessary data to model the package manager. In Cargo's case, for example, constraints would be extended to contain both version range constraints similar to Fig. 4.3a and a set of features that are required to be enabled in the dependency. Based on how the sets of versions and constraints have been chosen, the constraint satisfaction function (*sat*) needs to be adapted to model the package manager.

NPM's semantics for matching prerelease versions is subtle. A prerelease version can only satisfy a constraint if a sub-term of the constraint with the same semver version also has a prerelease. For example, consider the following constraint (spaces indicate a conjunction):

```
>1.2.3-alpha.3 <1.5.2-alpha.8
```

As expected, both `v1.2.3-alpha.7` and `v1.5.2-alpha.6` satisfy the constraint, since each is newer than `1.2.3-alpha.3` and older than `1.5.2-alpha.8`. However, the version `v1.3.4-alpha.7` *does not* satisfy the constraint, because while it respects the ordering, it is excluded as a non-explicitely mentioned prerelease version. This behavior prevents

the unexpected outcome of Maven's range constraints (Section 4.1.1). As part of our implementation of MAXNPM (Chapter 5) we have encoded these semantics in PACSOLVE, which works by having the *sat* function recursively check for constraint terms which contain identical semver versions with prereleases.

The semantics of Cargo's features requires a node in the solution graph to enable all the features which dependents requested, and no more. Specifically, Cargo implements a unification semantics, where the enabled features of a node N should be the union of the requested features from all predecessors of N in the solution graph. These semantics can be encoded in PACSOLVE by making use of both *sat* and *minGoal*. The *sat* function checks that the enabled features are a superset of the requested features, and the *minGoal* function would minimize at the highest priority the number of enabled features, so that no extras are included.

4.3.2 *Consistency and Cycles*

NPM, PIP, and Cargo all make different choices of consistency semantics, at different points along a spectrum of consistency strictness. PIP is the strictest of these three package managers: it forbids installation of different versions of the same package. This behavior is dictated by two factors. First, Python's module system identifies modules only by name, and if any two modules with the same name are in Python's module search path, one will always shadow the other. Second, at least when installing modules into a *single* Python installation or virtual environment, two modules with the same name will conflict on the filesystem.

Neither Rust nor JavaScript have Python's single version restriction; they allow installation of different versions of the same package. Node supports this because packages (JavaScript source files) are laid out in a `node_modules` *directory tree* on disk. Any directory in the tree can have its own local version of a dependency. Multi-versioning is possible in Rust because Rust binaries are statically linked, and rely on symbol rewriting to differentiate two instances of the same package. Cargo chooses to restrict this to only allow versions to be co-installed if they are not semver-compatible. So, under Cargo's semantics, `1.3.5` and `2.1.4` can be co-installed, but `1.3.5` and `1.4.2` cannot be co-installed. In the most relaxed case, NPM allows any versions whatsoever to be co-installed. All three of these consistency semantics can be encoded in PACSOLVE as shown in Fig. 4.5.

```
(define (npm-consistent v1 v2)
  #true)

(define (pip-consistent v1 v2)
  (equal? v1 v2))

(define (cargo-consistent v1 v2)
  (match `(,v1 ,v2)
    [`((0 0 ,z1) (0 0 ,z2))       #true]
    [`((0 ,y ,z1) (0 ,y ,z2))     (= z1 z2)]
    [`((0 ,y1 ,z1) (0 ,y2 ,z2))   #true]
    [`((x ,y1 ,z1) (x ,y2 ,z2))   (and (= y1 y2) (= z1 z2))]
    [_                            #true]))
```

Figure 4.5: Examples of three different consistency functions

In addition, the dependency solvers exhibit different semantics with regards to cycles: NPM and PIP allow for cyclic solution graphs, but Cargo forces all solution graphs to be acyclic.

4.3.3 *Properties of Dependency Solvers*

We have sketched the main differences between PIP, NPM, and Cargo, and shown how these differences can be encoded in PacSolve. We now ask the question: what are some properties that all dependency solvers *should* satisfy? We claim that the following properties are desirable:

1. (Soundness) The solution graph should satisfy all constraints;
2. (Completeness) The dependency solver should find a solution if one exists; and
3. (Optimality) The dependency solver should produce the lowest cost solution.

PacSolve makes it possible to formally state these properties:

Definition 4.3.1 (Soundness)**.** A dependency solver specified by $\mathcal{F}$ is *sound* if for all possible package metadata ($\mathcal{M}$) and all solutions that it produces (graphs G), we have $(\mathcal{F}, \mathcal{M}, G) \in \mathcal{S}$.

In our testing, all three of PIP, NPM and Cargo returned correct solution graphs, for appropriate choices of PacSolve semantics as outlined above. Note that this is a stronger property for dependency solvers with strict notions of consistency (PIP, Cargo), and for dependency solvers which disallow cycles (Cargo). We conjecture that PIP, NPM and Cargo all have sound dependency solvers.

UNSOUNDNESS OF MAVEN AND NUGET However, not all dependency solvers are sound. Maven and NuGet are like PIP, where each package must resolve to exactly one version. Their version constraints are also similar (and simpler) than PIP, NPM, and Cargo, so are easy to encode in PacSolve. However, they do not guarantee that their solutions satisfy all package constraints. Instead, when version consistency conflicts arise, they ignore all but the closest package constraints to the root. Thus Maven and NuGet are unsound with respect to a standard encoding of constraint semantics into PacSolve. An alternative encoding could define the *sat* to be the constant true function (so all constraints are trivially satisfied), and then include cost terms in *minGoal* if a constraint is not satisfied (effectively implementing soft constraints). Under this encoding, it may be possible that Maven and NuGet are sound. Regardless, PacSolve as a formal tool has helped to illuminate that constraints in Maven and NuGet work fundamentally differently than in many other dependency solvers.

Definition 4.3.2 (Completeness)**.** A dependency solver specified by $\mathcal{F}$ is *complete* if for all $\mathcal{M}$, if there exists a G where $(\mathcal{F}, \mathcal{M}, G) \in \mathcal{S}$, then the dependency solver produces a solution G' such that $(\mathcal{F}, \mathcal{M}, G') \in \mathcal{S}$.

Note that in this definition the only relationship between G and G' is that both are satisfying solution graphs, as this defintion simply means that if some satisfying dependency solution exists, then the solver is always capable of finding a solution, but there may multiple satisfying solutions.

In our testing, PIP always managed to successfully return a solution graph, if a solution graph exists according to PIP's semantics. PIP backtracks upon finding either conflicting nodes, or version constraints which cannot be satisfied by any package versions. Enough backtracking will eventually find a solution, if one exists (even if that solution is far from optimal, as explained below). We conjecture that PIP has a complete dependency solver. However, NPM and Cargo demonstrate different ways in how completeness can go wrong.

INCOMPLETENESS OF NPM NPM implements a non-backtracking dependency solver. While NPM cannot fail due to conflicts among version nodes, failure may occur when a version constraint cannot be satisfied by any possible version. In this case, NPM is unable to backtrack, and gives up, whereas PIP or Cargo can attempt finding other solutions. Consider a package `A` with two versions: `Av1.0.0` which has no dependencies, and `Av2.0.0` which depends on exactly version `Bv9.9.9`. Furthermore, suppose that `Bv9.9.9` does not exist in the package repository. (This situation happens in practice.) Now, suppose the root node then depends on any version of package `A`.

There exists a satisfying solution graph: the root node chooses `Av1.0.0`, which then has no further dependencies. This is the solution that Cargo and PIP will find, after they both attempt and backtrack away from `Av2.0.0`. However, NPM fails: it greedily commits to `Av2.0.0` and then fails to find `Bv9.9.9`.

INCOMPLETENESS OF CARGO Unlike NPM, Cargo performs backtracking similarly to PIP, so one might expect Cargo to have a complete solver. However, Cargo also checks that the solution graph is acyclic, but does not backtrack when a cycle is found. For example, suppose we have package `A` with two versions, and a package `B` with one version, with the following dependencies: `Av1.0.0` has no dependencies, `Av2.0.0` depends on any version of `B`, and `Bv1.0.0` depends on any version of `A`. Finally, the root node depends on any version of `A`. There exists an acyclic solution: the root node chooses `Av1.0.0`, which then has no further dependencies. However, in this situation Cargo will find a cyclic solution first, recognize that it is cyclic, and fail with an error.

Definition 4.3.3 (Optimality)**.** A dependency solver specified by $\mathcal{F}$ is *optimal* if for all $\mathcal{M}$, for any correct solution G that it produces given $\mathcal{M}$, there does not exist a G' with $(\mathcal{F}, \mathcal{M}, G') \in \mathcal{S}$ where G' has lower cost than G, that is, $\mathit{minGoal}(G') < \mathit{minGoal}(G)$.

The property of optimality depends on the specific choice of minimization criteria (*minGoal*), in addition to the other choices of semantics. A common optimization objective is to prefer newer versions of packages when possible, but NPM, PIP and Cargo do not agree on what "newer" means: PIP and Cargo prefer numerically larger version numbers, while NPM has a more complex strategy of preferring the temporally most recently uploaded version and then falling back to largest version numbers. More generally, a wide range of other optimization criteria are possible, as explained in Section 4.2.3. NPM, PIP and Cargo all apply their preference for newer versions (for their choice of newer) in a heuristic manner, guided by the order in which they explore solution graphs. All of these package managers are not, in general, optimal.

Table 4.1: Comparison of conjectured formal properties across package managers

Property	PIP	NPM	Cargo	Maven	PacSolve	MaxNPM (Chapter 5)
Soundness	✓	✓	✓	✗	✓	✓
Completeness	✓	✗	✗	n/a[3]	✓	✓[2]
Optimality	✗	✗	✗	n/a[3]	✓	✓[2]
Linking Soundness	✓	✓	✓	✓	n/a	✓
Linking Isomorphism	✗	✗[1]	✓	✗	n/a	✓

[1] Using NPM's default installation strategy (`hoisted`). The underlying Node module system supports linking isomorphism, which other JavaScript package managers (e.g. `pnpm`) and NPM in other configurations leverage.
[2] Up to implementing out-of-scope NPM features
[3] Our completeness and optimality definitions do not naturally apply when a solver lacks soundness.

4.3.4 *Semantics of Dependency Solvers in Relation to Semantics of Module Systems*

Dependency solvers do not operate as fully independent pieces of software. Rather, the dependency solutions they build are to be consumed by other software: in the case of programming language package managers, the solutions are consumed by the build system or language runtime. While a detailed account of the semantics of module systems [45, 57] is beyond the scope of this work, we do aim to give a brief idea of how to connect the semantics of dependency solvers with those of module systems, and what properties might be expected to hold.

In the PacSolve semantics, a solution graph $\langle N_R, D_R \rangle$ is an internal artifact of the dependency solver, and to be usable must be arranged into some data structure that is understood by the relevant module system. For example, NPM builds an in-memory dependency solution, but then must write to disk a `node_modules/` directory tree in a format understood by the Node runtime. Once the module system can load the dependency solution, the module system is then responsible for *resolving imports*, that is, deciding which module a given module name refers to. Here, we do not model all the intricacies of module systems that may happen *within* a package (nested modules, access modifiers, etc.), but only model the top-level modules defined by the dependency graph, and the importability relation between them.

We model how a module system behaves at resolving imports, given a solution graph $G = \langle N_R, D_R \rangle$, via a relation $\text{Imports}_{\langle N_R, D_R \rangle}$ between solved dependencies (and the **root**), and other solved dependencies, which is parameterized by the solution graph. Specifically, we define, for all $x \in N_R \cup \{\textbf{root}\}$ and $(p, v) \in N_R$, that $\langle x, (p, v) \rangle \in \text{Imports}_{\langle N_R, D_R \rangle}$ iff code in the package (or **root**) x is able to import definitions contained in solved dependency (p, v). We do not define the name through which the import is resolved, as many systems (such as PIP/Python and Cabal/Haskell), allow module names to be different from package names. The pair $\langle x, (p, v) \rangle \notin \text{Imports}_{\langle N_R, D_R \rangle}$ exactly when (p, v) can not be imported within the context of x.

Equivalently, we may view this as a directed graph, where nodes are solution graph nodes (and **root**) and edges exist when one solved dependency can import another ($x \to (p, v)$). Throughout the rest of this discussion we call this the *induced import graph*. Additionally, note that this construction allows there to be multiple outgoing edges from x to different versions of the same package (e.g. (p, v_1) and (p, v_2)), indicating that x is able to import multiple versions of the same package. While this may seem strange, it is in fact necessary to fully model the capabilities of some package managers, such as Cargo and NPM, which enable a programmer to specify aliased dependency names and then import different versions of the same package. Again, it is outside the scope of these semantics to describe how the module system decides which version to import, just that they are importable in some way.

It is important to remark that the induced import graph models more than just the semantics of the module system in isolation; it is also modeling the translation between the solver's solution graph and the data structure understood by the module system. Finally, it says nothing about what actually occurs at runtime: just because there is an edge $x \to (p, v)$ does not mean that x actually performs that import, or that the import wouldn't fail for other reasons (e.g. circular imports), just that the module system can resolve the import to (p, v).

We can now describe two key properties regarding this interplay between the behavior of the dependency solver and the module system:

Definition 4.3.4 (Linking Soundness)**.** A dependency solver and a module system *link soundly* if all solutions that the solver produces (graphs $G = \langle N_R, D_R \rangle$) are a spanning subgraph of its induced import graph.

Linking soundness is a property that any practical package manager should support, as it merely states that if the dependency solver produces an edge between dependencies x and y, then in fact y is importable from x. As a practical example, consider an NPM

package p which depends on three other packages, a, b and c, all of which have no dependencies themselves. The dependency solution graph that the NPM solver produces is a tree, with p as the root of the tree and (concrete versions of) a, b and c as children. However, due to how the Node runtime resolves imports, not only can p import a, b and c, but any of a, b and c can import another. The induced import graph looks like a fully connected graph between a, b and c, paired with the node p which has edges to the other three nodes. Here we can see that the tree-shaped solution graph is a spanning subgraph of the induced import graph, which means that if there is an edge between nodes within the dependency solver representation, then there must be an importable relationship between those nodes (and in the case of NPM, extra edges too). Linking isomorphism (below) is a strong property which excludes extra edges.

When combined with dependency solver soundness (Definition 4.3.1), linking soundness yields the important property that when a programmer specifies a constraint on a dependency, they will be guaranteed to be able to import that dependency with a version satisfying their constraint (unless solving fails). In our testing, all three of PIP, NPM and Cargo exhibited the conjunction of soundness and linking soundness, and we conjecture that all three package managers are both sound and linking sound with their respective language module systems.

However, linking soundness allows there to be additional edges in the induced import graph which do not exist in the solution graph (G). Our next definition of *linking isomorphism* captures this:

Definition 4.3.5 (Linking Isomorphism)**.** A dependency solver and a module system *link isomorphically* if all solutions that the solver produces (graphs $G = \langle N_R, D_R \rangle$) are identical to the induced import graph, that is, the identity function $\mathrm{id} : N_R \to N_R$ is a graph isomorphism between G and the induced import graph.

Linking isomorphism is a stronger property than linking soundness, and states that a solved dependency y is importable from x if and only if the dependency solver produces an edge between x and y. When composing this property with solver soundness, this yields the property (assuming solving succeeds) that a programmer can import a dependency if and only if they explicitly specified that dependency (and that they will receive a correct version).

Among our testing of PIP, NPM and Cargo, the only package manager to display the linking isomorphism property is Cargo. When performing solves, Cargo records not only which dependencies it selected, but also a directed acyclic graph structure corresponding to our notion of induced import graphs. At build time, the build

system component of Cargo reads this data structure to invoke the Rust compiler with arguments allowing it to import precisely the declared dependencies for a particular package.

NON-ISOMORPHIC LINKING IN PIP AND NPM Both PIP and NPM link soundly but non-isomorphically with the Python and Node runtimes, respectively. That is, programmers can write import statements that successfully import other packages, despite not declaring those packages as dependencies. Similarly to Cargo, NPM builds a detailed dependency tree structure both in-memory and on disk, keeping track of the dependencies of each package. However, NPM then performs an operation known as *hoisting*, which attempts to move all nodes up to the root of the tree as long as there is no conflict between multiple versions of the same package. When a package is moved to the root of the tree, it becomes importable by any other module. This property of NPM allows subtle dependency specification bugs to be introduced into programs, as programmers may accidentally start using transitive dependencies which are importable (and even autocompleted!) but are not specified in their `package.json`, and as such may be removed at any time. Non-isomorphic linking essentially prevents information hiding of dependency management decisions. One of the motivations of the `pnpm` package manager as a replacement for NPM was to make a different design decision to not perform hoisting [85], and thus `pnpm` satisfies isomorphic linking (which is born out by our manual testing). Additionally, NPM allows configuration of its installation strategy into three other modes [18], two of which we suspect satisfy isomorphic linking but we have not thoroughly tested. While the linking properties are not applicable to executable PACSOLVE itself (since it is only a dependency solver), when we build on PACSOLVE to implement MAXNPM in Chapter 5, we use a similar design of `pnpm` and produce `node_modules/` trees that are both compatible with Node and (we believe) satisfy isomorphic linking.

Unlike NPM, PIP does not eschew isomorphic linking out of an apparent design decision, but rather is forced to by the underlying data structure that the Python module system reads. All Python modules from all installed packages are placed in the global (per virtual environment) `site-packages/` directory in a flat structure identified by module name only. Any dependency graph structure must be forgotten, and only a set model of the installed dependencies remains. This is only possible because PIP disallows different versions of the same package (Section 4.3.2). From there, the only reasonable import graph semantics that could be induced by the set are to allow

any package to import any package. Indeed, PIP always produces a fully-connected importability graph between all installed packages[2].

In the context of programming languages PIP's behavior may appear lacking, but is reflective of language package managers having their historical roots in operating system package managers, for which this global set semantics is standard (e.g. all libraries are placed in `/usr/lib`, or a handful of other directories).

4.4 SYNTHESIZING SOLUTION GRAPHS WITH PACSOLVE

Performing dependency solving in the face of potentially conflicting dependencies is known to be NP-complete [27]. Some package managers use polynomial-time algorithms by giving up on various properties, such as disregarding conflicts (NPM) and eschewing completeness (NPM and PIP's old solver [89]). Since the PACSOLVE model includes a generalized notion of conflicts (*consistent*), we make use of SMT solvers to implement PACSOLVE efficiently.

We implement PACSOLVE in Rosette [100], which is a solver-aided language that facilitates building verification and synthesis tools for DSLs. In the PACSOLVE DSL, *the program is a solution graph*. We implement a function that consumes a) package metadata, b) a dependency solver specification (Fig. 4.2) and c) a solution graph, and then asserts that the solution graph is correct. Since we use Rosette, with a little effort, we can feed the predicate a *solution graph sketch* instead of a concrete solution graph. This allows us to use Rosette to perform *angelic execution* [10] to synthesize a solution graph that satisfies the correctness criteria. This section describes the synthesis procedure in more detail, starting with how we build a sketch.

SKETCHING SOLUTION GRAPHS Before invoking the Rosette solver, we build a sketch of a solution graph that has a node for every version of every package that is reachable from the set of root dependencies. Every node in a sketch has the following fields: a) a concrete name and version for the package that it represents (or the distinguished root node); b) a symbolic boolean *included* that indicates whether or not the node is included in the solution graph; c) a symbolic natural number *depth* which we use to enforce acyclic solutions when desired; d) a vector of concrete dependency

2 Actually, a more rigorous analysis would note that the structure is rather that of a pointed set and pointed graph, with **root** being the distinguished element. The importability graph is a fully connected graph on the set of installed packages but excluding **root**, along with an edge from **root** to every installed package vertex.

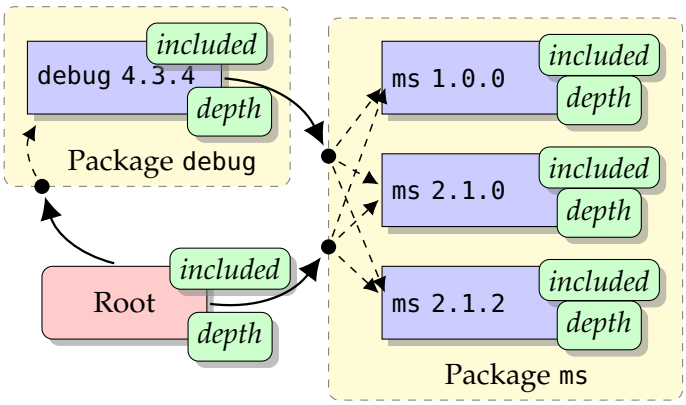


Figure 4.6: A sample solution graph sketch

package names; e) a vector of concrete version constraints for each dependency; and f) a vector of symbolic *resolved versions* for each dependency.

Fig. 4.6 illustrates the solution graph sketch corresponding to the dependency solving problem given in Fig. 4.1. The combination of concrete dependency names and symbolic dependency versions can be seen as representing symbolic edges in two parts: a concrete part which does not need to be solved for (solid arrows), and a symbolic part which requires solving (dashed arrows). This representation shrinks the solution space of graphs as outgoing edges can only point to nodes with the correct package name.

GRAPH SKETCH SOLVING We define three assertion functions that check correctness criteria of a solution graph:

1. `check-dependencies` consumes a graph and a node, and asserts that if the node is *included*, then all the dependencies of the node are a) *included* in the graph; and b) consistent with the associated version constraints as judged by the constraint interpretation function (*sat*). We run this assertion function on all nodes, and additionally assert that the root node is *included*.

2. `globally-consistent?` consumes a graph and asserts that for all pairs of nodes that are a) included in the graph, and b) have the same package name, their package versions are allowed to be co-installed as given by the consistency function (*consistent*).

3. `acyclic?` consumes a graph and a node, and asserts that the *depth* of the node is strictly less than the depth of all its dependencies. If an acyclic solution is desired, we run this assertion function on all nodes, and assert that the root node has depth zero.

We also use Rosette to execute the objective function (*minGoal*) on the graph sketch, yielding a symbolic real-valued formula, and then instruct Rosette to minimize it when concretizing the solution graph sketch. As a final step, to produce a solution graph, we traverse the concretized solution graph sketch from the root node, and produce those nodes that are marked *included*.

4.5 DISCUSSION

PacSolve is an expressive and powerful framework in which to formally describe the semantics of a dependency solver. In our design of PacSolve, we aimed to model the essential features and differences between many contemporary package managers, and as shown later in Chapters 5 and 6, PacSolve can be used as an effective foundation for building tools to support improved dependency optimization and repair across two different package ecosystems (NPM and Pip).

Since the designs of package managers can vary so widely, it is worth taking a step back to discuss the variety of features supporoted by some popular package managers. We reflect on the usefulness of these features, and whether or not they can be encoded in PacSolve.

4.5.1 *Multiple Dependency Versions & Conflicts*

As has already been discussed, some package managers, including NPM and Cargo, allow for multiple versions of the same package to be installed within one dependency solution. Additionally, both NPM and Cargo support dependency aliases, which allows *one package* to depend on multiple versions of the same package. PacSolve directly supports modeling these features.

Co-installation of multiple versions of a dependency has both upsides and downsides. It makes the solver strictly more relaxed, in the sense that more solutions are admitted under the semantics presented thus far, which practically means that programmers will face unsatisfiable dependency constraints less often. More specifically, this design allows programmers to incrementally upgrade dependencies: they may run into cases

where they would like to upgrade a (direct) dependency, but that same dependency is also transitively dependend on elsewhere in the graph in such a way that disallows the upgrade. Multiple versions allows the programmer to simply upgrade only their copy of the dependency, and leave the transitively dependency upon version in an old state.

While this allows for more flexibility in dependency management, there are a few downsides. First, multiple versions of packages bloats code size. While NPM and Cargo's solving algorithms attempted to unify dependencies, they do not do so optimally (Chapter 5), leading to increased code size in practice. Second, while programmers may not face unsatisfiable constraints up front, they are nevertheless subject to bugs or other problems that can be caused by having multiple versions of a package installed.

In Chapter 5 we use PacSolve to build a configurable replacement for NPM, and observe how many of the top 1000 most downloaded packages on NPM have unsatisfiable dependencies if multiple dependency versions are disallowed. We find that requiring multiple versions of a dependency is relatively rare (1.9% of the top 1000 packages), and thus suggest that this feature of NPM (and to a lesser degree Cargo) may not be worth its considerable downsides. Additional work including programmer surveys could be conducted to further examine its usefulenss.

A closely related topic is the notion of cross-package conflicts, which PacSolve does not support, nor do language-specific package managers, to the best of our knowledge. System package managers, in particular APT/dpkg do support this notion, in which a developer of package `foo` may declare that `foo` conflicts (and thus cannot be co-installed with) some other package `bar`. This feature may be both useful and reasonable for operating system level package management, as it is important for system stability to enforce that e.g. only one driver for a piece of hardware can be installed at once. At the scale of a programming language repository however, allowing programmers to specify arbitrary conflicts would harm composability and thus predicability of dependency solving, without a clear need for this feature. Two disjoint solution subgraphs may be simultaneously unsatisfiable, and it may be unclear to the programmer why that is the case.

4.5.2 *Dependency Overrides*

Both NPM and Cargo allow for a programmer to *override* dependencies, in which a package may forcefully select a different version for one of its transitive dependencies,

even one that would not satisfy the original version constraint. While not intended to be used frequently, this may be an important feature to enable programmers to update transitive when a package in the middle did not specify a flexible version constraint, particularly in crisis situations such as rapid responses to security vulnerabilities.

PacSolve does not spefically model dependency overrides, and in fact it is unclear what a reasonable semantic model for dependency overrides should be. NPM and Cargo vary widely in their design, each offering a different configuration language to specify paths through the solution graph that should be overridden. Additionally, it is unclear if overrides from the root package only should apply to the solution, or overrides from any package in the solution: Cargo only considers root package overrides, while NPM does not specify its behavior. More research could be done on the design of dependency override systems, and what design serves a good balance of solving practical problems faced by programmers and maintaining predictable solving semantics.

4.5.3 *Dependency Staging*

One of the most notable features of many language package managers that PacSolve does not explicitely model is a notion of stages of dependency solving, also called scoped dependencies. Many package managers, including Maven, NPM, Cargo and others allow programmers to, for example, specify dependencies for their tests, or development environment, separately from runtime dependencies. For interpreted languages such as NPM, this is not a barrier to using PacSolve in practice, as one can easily produce a "test only" dependency subgraph, and solve that with PacSolve. By convention, TypeScript source code is not submitted to the NPM ecosystem (compiled JavaScript is), so resolving a dependency on the TypeScript compiler does not have to be done first in order to install dependencies correctly.

However, the situation is more interesting when compilation or macro systems are involved. Rust features a robust macro system, including procedural macros which are arbitrary pieces of Rust code that operate on the Rust AST at compile time. Cargo must first solve dependencies for any procedural macro binaries, then execute the procedural macros to obtain the expanded source code and finally compile it. Importantly, any dependencies that the procedural macro uses do not need to be included (i.e. linked) at runtime of the final binary, only during runtime of the macro. However, it remains unclear how dependency unification happens or should happen across stages, and

exploring these details at the semantic level would require expanding the PacSolve model.

4.5.4 *Virtual Packages*

As with cross-package conflicts, the APT/dpkg system package managers also support a cross-package relationship of virtual packages, which allow multiple different packages to provide a single named interface, and a consumer may depend on any packages which provides that interface. While this may be useful for system level package management (for example multiple competing `awk` implementations), the utility in language level package management remains unclear. Certain use cases may exist, such as multiple different database connector libraries, but it is unclear if there are sufficient practical use cases to justify additional solver complexity to handle this feature.

Modeling virtual packages is possible but not practical in PacSolve: because package names in dependencies are fixed (non-symbolic), in order to have the solver solve for which concrete package should be selected, one would need to move the package name into the (symbolic) version datatype, which would then allow it to be selected freely by the solver. However, this would increase the size of the solution space and may perform poorly without additional optimization work.

4.6 CONCLUSION

We present PacSolve, a semantics of dependency solving that we use to highlight the essential features and variation within the package manager design space. Next, we use PacSolve to implement MaxNPM, a drop-in replacement for NPM that allows the user to customize dependency solving with a variety of global objectives and consistency criteria.

5

MAXNPM: FLEXIBLE AND OPTIMAL DEPENDENCY MANAGEMENT FOR JAVASCRIPT VIA PACSOLVE

Package managers such as NPM (the de facto package manager for JavaScript) have become essential for software development. For example, the NPM repository hosts over two million packages and serves over 43 billion downloads weekly. The core of a package manager is its *dependency solver*, and NPM's solver tries to quickly find dependencies that are recent and satisfy all version constraints. Unfortunately, NPM uses a greedy algorithm that can duplicate dependencies and can even fail to find the most recent versions of dependencies.

Moreover, the users of NPM may have other goals that NPM does not serve. a) Web developers care about minimizing code size to reduce page load times. "Bundlers" such as Webpack alter the packages selected by NPM to eliminate duplicates (Section 5.1.2). b) Many developers want to avoid vulnerable dependencies and several tools detect and update vulnerable dependencies, including NPM's built-in "audit" command [73]. However, the audit command is also greedy and its fixes can introduce more severe vulnerabilities (Section 5.1.1). c) Finally, there are semantic reasons why many packages, such as frameworks with internal state, should never have multiple versions installed simultaneously. However, NPM's approach to solving this, known as "peer dependencies" is brittle and causes confusion (Section 5.1.3).

The problem with these approaches is that they are ad hoc attempts to customize and workaround NPM's solver. Bundlers and audit tools modify solved dependencies after NPM produces its solution. Peer dependencies effectively disable the solver in certain cases and rely on the developer to select unsolved dependencies at the package root. In general, NPM cannot produce a "one-size-fits-all" solution that satisfies the variety of goals that developers have. Moreover, any tool that modifies the solver's solution after solving risks introducing other problems and may not compose with other tools.

Our key insight is that all these problems can be framed as instances of a more general problem: optimal dependency solving, where the choices of optimization objectives and constraints determine which goals are prioritized. Due to the wide-ranging goals in the NPM ecosystem, we argue that NPM should allow developers to *customize and combine several objectives*. For example, a developer should be able to specify policies such as

"dependencies must not have any critical vulnerabilities", "packages should not be duplicated", and combine these with the basic objective of "select the latest package versions that satisfy all constraints." To make this possible and evaluate its effectiveness, we present MaxNPM: a complete, drop-in replacement for NPM, which empowers developers to combine multiple objectives. The heart of MaxNPM is a dependency solver for NPM built using PacSolve (Chapter 4).

We use MaxNPM to conduct an empirical evaluation with a large dataset of widely-used packages from the NPM repository. Our evaluation shows that MaxNPM outperforms NPM in several ways:

1. chooses newer dependencies compared to using NPM for 14% of packages with at least one dependency.
2. shrinks the footprint of 21% of packages with at least one dependency.
3. reduces the number or severity of security vulnerabilities in 33% of packages with at least one dependency.

Overall, MaxNPM takes just 2.6s longer than NPM on average, though encounters some outliers which solve significantly more slowly with MaxNPM, which is reflected in the standard deviation of the slowdown (13.7s).

5.1 BACKGROUND ON WORKING WITH NPM

NPM is the most widely used package manager for JavaScript. NPM is co-designed with Node, which is a popular JavaScript runtime for desktop and server applications. However, NPM is also widely used to manage web applications' dependencies, using "bundlers" like Webpack to build programs for the browser.

As discussed in Chapter 4, an unusual feature of NPM is that it may select multiple versions of the same package. Unfortunately, NPM's behavior is not always desirable, and can lead to increased code size and subtle runtime bugs. Moreover, NPM does not guarantee that packages are only duplicated when strictly necessary.

5.1.1 *Avoiding Vulnerable Dependencies*

NPM's built-in tool `npm audit` checks for vulnerable dependencies by querying the GitHub Advisory Database. The tool can also fix vulnerabilities by upgrading dependen-

cies without violating version constraints.[1] However, the tool has several shortcomings. a) Each run only tries to fix a single vulnerability. b) It only upgrades vulnerable dependencies, even if a vulnerability-free downgrade is available that respects version constraints. c) It does not prioritize fixes by vulnerability scores (CVSS), even though they are available in the GitHub Advisory Database. d) It does not make severity-based compromises. For example, a fix may introduce new vulnerabilities that are more severe than the original.

5.1.2 *Minimizing Code Bloat*

A "bundler", such as Webpack, Browserify, or Parcel, is a tool that works in concert with NPM to manage the dependencies of front-end web applications. The primary task of a bundler is to package all dependencies to be loaded over the web, instead of the local filesystem. However, bundlers do more, including work to minimize page load times. A simple way to minimize page load times is to reduce code size. Unfortunately, NPM's willingness to duplicate packages can lead to increased code size [91]. Contemporary bundlers employ a variety of techniques from minification to unifying individual files with identical contents. However, these techniques are not always sound and have been known to break widely-used front-end frameworks [71, 103].

5.1.3 *Managing Stateful Dependencies*

NPM's ability to select several versions of the same dependency is also unhelpful when using certain stateful frameworks. For example, React is a popular web framework that relies on internal global state to schedule view updates. If a program depends on two packages that transitively depend on two different versions of React, it is likely to encounter runtime errors or silent failures. The only way to avoid this problem is if all package authors are careful to mark their dependency on React as a *peer dependency*: a dependency that is installed by some other package in a project. However, there is no easy way to determine that all third-party dependencies use peer dependencies correctly. It can also be hard to determine before hand that a package will never be used as a dependency and thus should not use peer dependencies.

1 The `--force` flag breaks constraints and potentially breaks the program.

5.2 THE INTERFACE OF MAXNPM

The goal of MaxNPM is to help developers address the broad range of problems described above. MaxNPM serves as a drop-in replacement for the default `npm install` command. The user can run `npm install --maxnpm` to use MaxNPM's customizable dependency solver instead. There are two broad ways to customize MaxNPM. First, MaxNPM allows the user to specify a prioritized list of objectives with the `-minimize` flag. Out of the box, MaxNPM supports the following objectives (defined precisely in Fig. 4.4):

- `min_oldness`: minimizes the number and severity of installed old versions;
- `min_num_deps`: minimizes the number of installed dependencies;
- `min_duplicates`: minimizes the number of co-installed different versions of the same package; and
- `min_cve`: minimizes the number and severity of known vulnerabilities.

Second, MaxNPM allows the user to customize how multiple package versions are handled with the `-consistency` flag:

- `npm`: the default behavior of NPM, which allows several versions of a package to co-exist in a single project; and
- `no-dups`: require every package to have only one version installed.

With some work, the user can even define new objectives and consistency criteria.

For example, a developer building a front-end web application may want to reduce code size and select recent package versions. They could use MaxNPM as follows:

```
npm install --maxnpm --consistency no-dups
  --minimize min_oldness,min_num_deps
```

This command avoids duplicating dependencies, and minimizes oldness and the number of dependencies in that order. This is a more principled approach to reducing code size than the ad hoc de-duplication techniques used by bundlers. Moreover, we show that this command is frequently successful at reducing code size (Section 5.4.1.3).

As a second example, consider a developer building a web application backend, where they are very concerned about security vulnerabilities. They could use MaxNPM as follows:

```
npm install --maxnpm --minimize min_cve,min_oldness
```

This command subsumes `npm audit fix` and we show that it is substantially more effective (Section 5.4.1.1).

5.3 BUILDING MAXNPM USING PACSOLVE

MaxNPM is a package manager for JavaScript built on PacSolve, which is a DSL for describing dependency solvers. To use PacSolve to implement a fully-fledged package manager, there are three main components to develop: a) Concrete choices of semantic properties, b) a procedure for building PacSolve queries based on dependencies and the package ecosystem, and c) a procedure to decode a PacSolve result into the correct structure on disk which is loadable by the module system (in this case, the Node runtime). We walk through these components in the case of MaxNPM.

SETTING CONCRETE SEMANTICS The concrete semantics that are specified in a PacSolve query are a declarative specification of the dependency solver's behavior. While PacSolve makes it easier to get these correct compared to an ad-hoc implementation, one must define them carefully. When using PacSolve to build a package manager to interoperate with an existing package manager (in the case of MaxNPM and NPM), the existing package manager's semantics provide some formal guidance on how to choose the semantics. Specifically, the constraint satisfaction semantics (sat) of the new package manager should align with the old package manager, though the domain of versions and constraints need not be the same between the two. If we assume the old package manager (not implemented, but modeled with PacSolve) has a domain of versions ($\mathcal{V}_{old}$), constraints ($\mathscr{C}_{old}$), and satisfaction predicate ($sat_{old} : \mathscr{C}_{old} \to \mathcal{V}_{old} \to \textbf{Bool}$), then the new package manager must know how to translate versions and constraints into its choice of new domains ($enc_v : \mathcal{V}_{old} \to \mathcal{V}$ and $enc_c : \mathscr{C}_{old} \to \mathscr{C}$), and define the semantics of constraint satisfaction ($sat : \mathscr{C} \to \mathcal{V} \to \textbf{Bool}$) such that satisfaction in one package manager is equivalent to satisfaction in the other:

$$\forall v_{old} \in \mathcal{V}_{old}, \forall c_{old} \in \mathscr{C}_{old},\ \ sat_{old}(c_{old}, v_{old}) \iff sat(enc_c(c_{old}), enc_v(v_{old}))$$

The motivation to choose new version and constraint domains to be different from old domains is to encode version and constraint information in datatypes that are solvable or more efficiently solvable by the PacSolve implementation. For example,

in NPM's implementation, versions are a 4-tuple $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \textbf{String}$, where the last component is the prerelease string (e.g. `alpha-3`). In the MaxNPM implementation we encode this to be a natural number induced by the lexicographic ordering of prerelease strings within a single PacSolve query, and likewise for prerelease strings contained inside constraints. This ensures that prerelease ordering will be translated to Z3's theory of bitvectors, and we can use the above reasoning to check that this translation is correct.

PREPARING QUERIES MaxNPM must decide at runtime what set of packages ($\mathcal{N}$) to include in the PacSolve query. For the solver to be complete and optimal (Section 4.3.3), MaxNPM must include enough nodes in $\mathcal{N}$ such that if there exists a solution, then an optimal solution can be found using only the nodes in $\mathcal{N}$. A sound algorithm for this is a graph traversal starting at the root node, and moving from node x to node $y = (p', v)$ if for any $(p, c) \in \mathit{deps}(x)$, $p = p'$ and $\mathit{sat}(c, v)$ is true. Any *potentially* useful node for the solve will be visited by the graph traversal. It can easily be seen that any correct and optimal solution graph would be a subgraph of the graph traversal. This query preparation is agnostic to the specific package manager, and would be applicable to other package managers targeting PacSolve.

DECODING THE RESULT PacSolve will return a directed multigraph as a solution back to the package manager when solving is complete. As a final step, the package manager must take the PacSolve solution graph and perform the actual package downloading and installation steps in order to place everything on disk in a format understood by the target language runtime (in this case, the Node runtime). As discussed in Section 4.3.4, there are interesting design decisions in this space, and at the time of this writing NPM even supports four distinct configurable modes for how it links dependency solutions with the Node runtime [18]. In MaxNPM, we elected to make the design decision of nesting dependencies within the corresponding parent's `node_modules/` directory, and using symlinks as necessary for shared dependencies. Unlike NPM (in the standard configuration), this design achieves isomorphic linking (Section 4.3.4), ensuring that a package can only import a dependency if it declares it explicitly.

5.4 EVALUATION

We evaluate MaxNPM in several scenarios, determining whether its support for flexible optimization objectives can provide tangible benefits to developers as compared to NPM. We gather two large datasets of popular packages, and also investigate if MaxNPM is sufficiently reliable and performant to use as a drop-in replacement for NPM. In this section we use MaxNPM to answer each of our research questions:

RQ1: Can MaxNPM find better solutions than NPM when given different optimization objectives?

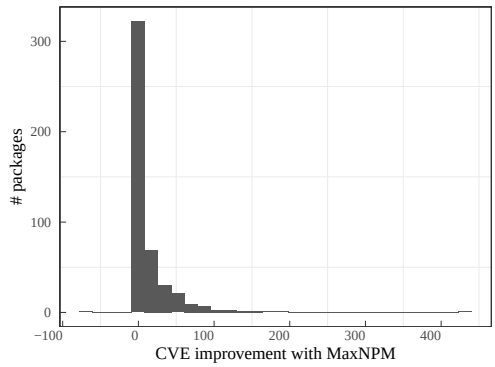
RQ2: Do MaxNPM's solutions pass existing test suites?

RQ3: Does MaxNPM successfully solve packages that NPM solves?

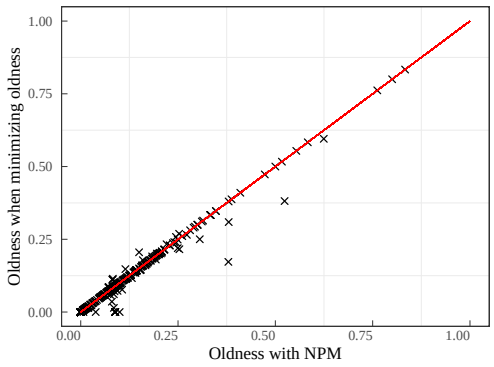
RQ4: Does using MaxNPM substantially increase solving time?

We build two datasets of NPM packages. *Top-1000* is a set of the latest versions of the top 1,000 most-downloaded packages as of August 2021. Including their dependencies, there are 1,147 packages in this set. Unsurprisingly, these packages are maintained and have few known vulnerabilities. Therefore, to evaluate vulnerability mitigation, we build the *Vuln-715* dataset of 715 packages with high CVSS scores as follows: a) we filter the *Top-1000* to only include packages with available GitHub repositories; b) we extract every revision of `package.json`; c) for each revision, we calculate the aggregate CVSS score of their direct dependencies, as determined by the GitHub Advisory Database; and d) we select the highest scoring revision of each package.

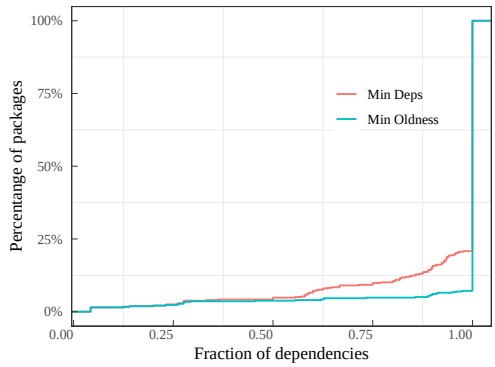
MaxNPM is built on NPM 7.20.1. PacSolve uses Racket 8.2, Rosette commit `1d042d1`, and Z3 commit `05ec77c`. We configure NPM to not run post-install scripts and not install optional dependencies. We run our performance benchmarks on Linux, with a 16-Core AMD EPYC 7282 CPU with 64 GB RAM. We warm the NPM local package cache before measuring running times.



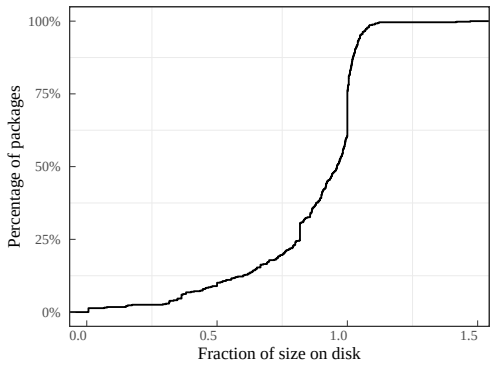
(a) A histogram showing CVSS improvement of packages solved with MaxNPM (configured to minimize vulnerabilities first) compared to NPM's auditing tool. CVSS decreases with 33% of packages.



(b) MaxNPM is configured to minimize oldness and the number of dependencies (in that order). Each point represents a package, and those below the line have newer dependencies, by the metric in Section 5.4.1.2. Overall, MaxNPM finds newer versions for dependencies.



(c) An ECDF of the fraction of dependencies compared to NPM, when MaxNPM is configured to a) minimize dependencies then oldness (red ECDF), or b) minimize oldness then dependencies (blue ECDF). MaxNPM can reduce dependencies in about 21% of packages.



(d) An ECDF of the ratio of disk space of packages solved using MaxNPM (configured to minimize number of dependencies, then oldness) vs NPM. MaxNPM can reduce space required.

Figure 5.1: Comparing NPM's to MaxNPM's solution quality. These plots ignore failures in both solvers and have MaxNPM configured to use NPM-style consistency and allow cycles.

5.4.1 *RQ1: Can* MaxNPM *find better solutions than NPM when given different optimization objectives?*

5.4.1.1 *Can* MaxNPM *help avoid vulnerable dependencies?*

We configure MaxNPM to minimize the aggregate CVSS scores of all dependencies,[2] and compare with the built-in `npm audit fix` tool (Section 5.1.1). We use the *Vuln-715* packages for this comparison. Both tools run successfully on 472 packages: the failures occur because these are typically older versions that do not successfully install.

The histogram in Fig. 5.1a reports the difference in aggregate CVSS score between `npm audit fix` and MaxNPM. A higher score indicates that a package has fewer vulnerabilities with MaxNPM. MaxNPM produces fewer vulnerabilities on 235 packages (33%). There is one package where MaxNPM produces a lower score; we are investigating this as a possible bug. The mean CVSS improvement by MaxNPM is 14.75 CVSS points (a "maximum severity" vulnerability is 10 points), or by 30.51%. The improvement is statistically significant ($p < 2.2 \times 10^{-16}$) using a paired Wilcoxon signed rank test, with a medium Cohen's d effect size of $d = 0.46$. Thus, we find that MaxNPM is substantially more effective than `npm audit fix` at removing vulnerable dependencies.

An example project where MaxNPM eliminates vulnerabilities is the `babel` compiler (34 million weekly downloads). Commit `5b09114b8` is in *Vuln-715*, and MaxNPM eliminates all vulnerabilities; whereas `npm audit fix` leaves several with an aggregate CVSS score of 59.4.

5.4.1.2 *Can* MaxNPM *find newer packages than NPM?*

MaxNPM ought to be able to find newer packages than NPM's greedy algorithm. We define the oldness of a dependency on a package version ($old(p, v)$) as a function that assigns the newest version the value 0, the oldest version the value 1, and other versions on a linear scale in between. We define the mean oldness of project as the mean oldness of all dependencies in a project,including transitive dependencies. Note that this metric is not identical to the minimization objective of MaxNPM, which calculates the sum and ignores duplicates. The metric is more natural to interpret, whereas the objective function avoids pathological solutions.

Figure 5.1b shows a point for every package in the *Top-1000*, with mean oldness using NPM and MaxNPM as its x and y coordinates. Points on $y = x$ are packages

2 The `min_cve,min_oldness` flags.

whose dependencies are just as old with both NPM and MaxNPM. 14% of packages excluding those with zero dependencies are better with MaxNPM, while 5% are worse. On average oldness improved by 2.62%. The improvement is statistically significant ($p = 4.27 \times 10^{-6}$) using a paired Wilcoxon signed rank test, with a small Cohen's d effect size of $d = 0.024$. Thus MaxNPM produces newer dependencies on average.

An example of successful oldness minimization is the `class-utils` package (15 million weekly downloads). MaxNPM chooses a slightly older version of a direct dependency, which allows it to chose much newer versions of transitive dependencies.

One might wonder why MaxNPM does worse in 5% of cases, since MaxNPM should be optimal. Manual investigation of these cases shows that some packages make use of features which we have not implemented in MaxNPM, such as URLs to tarballs rather named dependencies. MaxNPM is unable to explore that region of the search space. Implementing these features would take some engineering effort, but wouldn't require changes to the model.

5.4.1.3 *Can* MaxNPM *reduce code bloat?*

Instead of using ad hoc and potentially unsound techniques to reduce code bloat, we can configure MaxNPM to minimize the total number of dependencies. On the *Top-1000* packages, we configure MaxNPM in two ways: 1) prioritize fewer dependencies over lower oldness; and 2) prioritize lower oldness over fewer dependencies. Figure 5.1c plots an ECDF (empirical cumulative distribution function) plot where the x-axis shows the shrinkage in dependencies compared to NPM, and the y-axis shows the cumulative percentage of packages with that amount of shrinkage. The plot excludes packages with zero dependencies, and $x = 1$ indicates no shrinkage (when MaxNPM produces just as many dependencies as NPM).

Both configurations produce fewer dependencies than NPM, but prioritizing fewer dependencies is the most effective (the red line). For about 21% of packages, MaxNPM is able to reduce the number of dependencies, with an average reduction in the number of packages of 4.37%. The improvement is statistically significant ($p < 2.2 \times 10^{-16}$) using a paired Wilcoxon signed rank test, with a moderate Cohen's d effect size of $d = 0.20$. For the same set of packages, the total disk space required shrinks significantly (Fig. 5.1d). With MaxNPM, a quarter of the packages require 82% of their original disk space. Even when we prioritize lowering oldness, MaxNPM still produces fewer dependencies (the blue line).

An example of dependency size minimization is the `assert` package (13 million weekly downloads). For 3 direct dependencies, MaxNPM chooses slightly older revisions (with the same major and minor version). But this eliminates 33 of 43 transitive dependencies.

We have observed that in a few cases NPM exhibits a bug in which it installs additional dependencies that are not defined in the set of production dependencies of the package, nor in the set defined by transitive dependencies[3]. However, MaxNPM does not exhibit this bug. We will work on reporting this bug, but we believe this an example of the advantage of PacSolve's declarative style of building package managers.

5.4.1.4 *Can* MaxNPM *address duplicated packages?*

			Minimization Objectives			Failures		
Solver	Consistency	Allow cycles?	Primary	Secondary	Successes	Unsat	Timeout	Other
NPM					953	0	0	47
MaxNPM	npm	Yes	Oldness	#deps	972	0	27	1
MaxNPM	npm	Yes	#deps	Oldness	972	0	27	1
MaxNPM	npm	Yes	Oldness	Duplicate	973	0	26	1
MaxNPM	no-dups	Yes	Oldness	#deps	926	19	54	1
MaxNPM	npm	No	Oldness	#deps	972	0	27	1
MaxNPM	no-dups	No	Oldness	#deps	926	19	54	1

Table 5.1: Failures that occur when running NPM and different configurations of MaxNPM on the *Top-1000* dataset.

NPM happily allows a program to load several versions of the same package, which can lead to subtle bugs (Section 5.1.3). To address this problem, a developer can configure MaxNPM to disallow duplicates. In this configuration, 19 packages (1.9%) in *Top-1000* produce unsatisfiable constraints, which indicates that they require several versions of some package (Table 5.1).

For example, `terser@5.9.0` is a widely used JavaScript parser that that directly depends on `source-map@0.7.x` and `source-map-support@0.5.y`. However, the latter depends on `source-map@0.6.z`, thus the build must include both versions of `source-map`. The ideal fix would update `source-map-support` to support `source-map@0.7.x`.

3 `@babel/plugin-proposal-export-namespace-from` is an example.

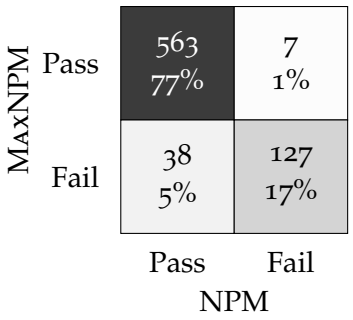


Figure 5.2: Results of running tests after solving dependencies with NPM and MaxNPM. In total only 5% of packages have a failing test with MaxNPM but not with NPM.

5.4.2 *RQ2: Do* MaxNPM*'s solutions pass existing test suites?*

Statistic	NPM Pass & MaxNPM Pass	NPM Pass & MaxNPM Fail
Mean	489.36	58.90
STD	4699.01	266.87
Minimum	0.00	0.00
25th Perc.	0.00	1.50
50th Perc.	7.00	4.00
75th Perc.	39.00	10.50
Maximum	81582.00	1493.00

Table 5.2: Statistics of the number of executed tests per package in the top left and top right groups of Fig. 5.2.

Although a package should pass its test suite with any set of dependencies that satisfy all constraints, in practice tests may fail with alternate solutions due to under-constrained dependencies. We identified test suites for 735 of the *Top-1000* packages. A test suite succeeds only if all tests pass. All test suites succeed with both MaxNPM and NPM on 77% of packages, and fail for both on 17%. There are 38 packages where

MaxNPM fails but NPM passes, and 7 packages where NPM fails and MaxNPM succeeds (Figure 5.2).

Test failures occurring slightly more often with MaxNPM's solutions are likely due to the fact that many of these packages have already been solved and tested with NPM, so even if their dependencies are underconstrained, at present NPM produces working solutions. Manual investigation suggests that the 7 packages that fail with NPM but succeed with MaxNPM are likely due to flaky tests and missing development dependencies while the 38 packages that fail with MaxNPM but succeed with NPM are due to those reasons in addition to under-constrained dependencies.

Finally, to verify that packages which pass their tests with MaxNPM are not doing so vacuously due to no or few tests, in Table 5.2 we report statistics of the number of executed tests for packages in the group that pass with NPM and MaxNPM, and in the group that pass with NPM and fail with MaxNPM. A two-sided Mann-Whitney U test indicates that there is no statistically significant difference between the two populations ($p = 0.49$).

5.4.3 *RQ3: Does* MaxNPM *successfully solve packages that NPM solves?*

The rightmost three columns of Table 5.1 show the number of failures resolving dependencies on the *Top-1000* for NPM, along with each configuration of MaxNPM that we evaluated. On the *Top-1000* packages, NPM itself fails on 47 packages. Many of these failures occur due to broken, optional peer-dependencies that MaxNPM does not needlessly solve.[4] We run MaxNPM in several configurations, and get 26—28 failures when *duplicate versions are permitted*. Some failures occur across all configurations, e.g., one package requires macOS. Most of our other failures are timeouts: we terminate Z3 after 10 minutes. When duplicates are not permitted, we do get more failures due to unsatisfiable constraints, but these are expected (Section 5.4.1.4). Some users may prefer to have MaxNPM fail when it cannot find a solution rather than falling back to an unconstrained solution, as the latter may lead to subtle and hard-to-debug issues at runtime due to e.g. conflicting global variables in multiple versions of the same package. When we permit duplicates like NPM, we find that MaxNPM successfully builds *more* packages than NPM itself, providing strong evidence that MaxNPM can reliably be used as a drop-in replacement for NPM.

4 They are not necessary to build, but NPM attempts to solve for them even with the `--omit-peer` flag.

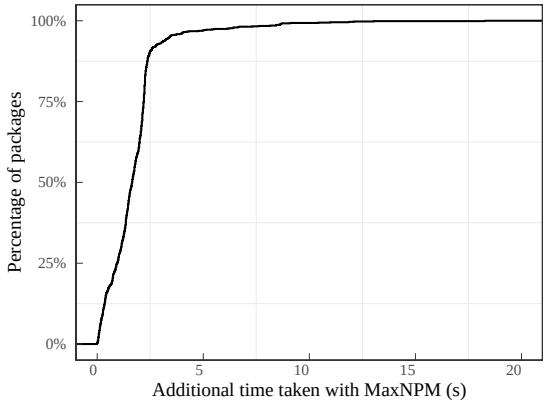


Figure 5.3: ECDF of the additional time taken by MaxNPM to solve and install packages compared to NPM, ignoring timeouts and failures, with outliers ($> 20s$) excluded. The outliers take up to 329s extra seconds, but the mean and median slowdowns are only 2.6s and 1.6s, respectively. In this experiment MaxNPM was configured with NPM-style consistency, allowing cycles, and minimizing oldness first and then number of dependencies.

5.4.4 *RQ4: Does using* MaxNPM *substantially increase solving time?*

On the *Top-1000* packages, we calculate how much *additional time* MaxNPM takes to solve dependencies over NPM. We observe that the minimum slowdown is $-2.3s$ (when MaxNPM is faster than NPM), the 1st quartile is 0.8s, the median is 1.6s, the mean is 2.6s, the 3rd quartile is 2.2s, the max is 329s, and the standard deviation of the slowdown is 13.7s. These absolute slowdowns are on top of the baseline of NPM, which takes 1.52s on average, and 1.34s at the median. We exclude timeouts from this analysis, we report those in Table 5.1. As evidenced by the maximum and standard deviation, there are a few outliers where MaxNPM takes substantially longer. We also perform a paired Wilcoxon signed rank test and find that the slowdown is statistically significant ($p < 2.2 \times 10^{-16}$), with a moderate Cohen's d effect size of $d = 0.27$. Figure 5.3 shows an ECDF of the absolute slowdown, but with outliers (> 20s) removed. We conclude that while MaxNPM does increase solving time, the increase is modest in the majority of cases, but there are a few outliers. This performance characteristic mirrors that of other SAT-solver based package managers, including production ones such as Conda [19].

Excluding outliers, a significant portion of the overhead is serializing data between JavaScript (MaxNPM) and Racket (PacSolve), which could be improved by building the solver in JavaScript or using a more efficient serialization protocol. As for the outliers, one could implement a tool that first tries MaxNPM but reverts to greedy solving after a timeout, at the expense of optimality.

5.5 DISCUSSION

By modeling NPM in PacSolve and comparing their real-life behavior, we gained valuable insight into NPM's behavior. As already explored empirically in Section 5.4, NPM is non-optimal, and it is challenging to see how it could be optimal without implementing a full solver based approach such as MaxNPM. However, NPM does have some lower-hanging fruit that is easier to achieve and would benefit users. First, as explored theoretically in Section 4.3 and empirically in Section 5.4 NPM is not in fact *complete*, in that there are situations where a satisfying solution exists, but NPM fails to find it. Most commonly, a version of a package depends on a dependency which does not exist, and NPM immediately bails out rather than backtracking. This could be implemented with simple backtracking without harming the performance of solves which currently succeed. In addition, Section 5.1.1 identifies several shortcoming of the `npm audit fix` tool at the time of our testing. We would suggest incorporating severity of vulnerabilities into the update logic, so that the tool can decide trade-offs between different vulnerabilities. The tool is also unable to downgrade dependencies to remove vulnerabilities, which would be a useful option to have, even if not enabled by default.

5.6 THREATS TO VALIDITY

EXTERNAL VALIDITY The projects that we used in our evaluation may not be representative of the entire ecosystem of NPM packages. We select the 1,000 most popular projects, and report performance as a distribution over this entire dataset, including a discussion of outliers. Given the number of projects that we used in our evaluation and their popularity, we believe that MaxNPM is quite likely to be helpful for improving package management in real-world scenarios. We describe PacSolve as a unifying framework for implementing dependency solvers, however we only use PacSolve to implement MaxNPM. Future work should empirically validate PacSolve's efficacy in other ecosystems.

INTERNAL VALIDITY MaxNPM, PacSolve, and the tools that we build upon may have bugs that impact our results. To verify that differences between NPM and MaxNPM are not due to bugs in MaxNPM, we carefully analyzed the cases where MaxNPM and NPM diverged in their solution, and we walkthrough some example cases in Section 5.4. Additionally, we have carefully written a suite of unit tests for PacSolve.

CONSTRUCT VALIDITY We evaluate MaxNPM's relative performance to NPM when optimizing for several different objective criteria. However, it is possible that these criteria are not meaningful to developers. For example, when comparing MaxNPM and `npm audit fix` in reducing vulnerabilities, we use the aggregate vulnerability scores (CVSS) to rank the tools. However, in practice, these scores may not directly capture the true severity of a vulnerable dependency in the context of a particular application. MaxNPM does however allow for potential customization of constraints to fit the developer's needs. Future work should involve user studies, observing the direct impact of PacSolve-based solvers (including MaxNPM, and implementations for other ecosystems) on developers.

5.7 DATA AVAILABILITY

Our artifact is available under a CC-BY-4.0 license [83] and consists of a) the implementations of PacSolve and MaxNPM, b) the *Top-1000* and *Vuln-715* datasets, and c) scripts to reproduce our results. All of our code is also available on GitHub [81], and MaxNPM can be easily installed with `npm install -g maxnpm`.

6

REPYRO: AUTOMATED DEPENDENCY REPAIR THROUGH CONSTRAINT MUTATION

Unfortunately, dependency solutions do not always work as intended, which may cause any number of software failures including build failures, linting failures, runtime crashes, runtime bugs, performance bugs, etc. Generally, this happens due to *dependency drift* with older software, where at the time of writing the dependency constraints produced a solution that worked correctly, but as new versions of dependencies were published these same constraints produced different solutions.

Programmers who now wish to use this older code face the challenge of debugging the dependency constraints. Generally, this consists of two subtasks: a) determining which dependencies are the root causes of the failures, and b) how to modify the constraints of bad dependencies to fix the failures. Along the way, programmers encounter other challenges, including hidden effects (changing one dependency version also changes many others) and unsatisfiable constraints (requiring backtracking on constraint modification decisions). Repairing dependencies is a difficult and time-consuming task.

In this chapter we propose Repyro, an automated Python dependency repair framework based on PacSolve that allows large-language models (LLMs) to be used to suggest fixes to PacSolve constraints, while PacSolve optimization objectives are used to prioritize more relevant solutions. Repyro extends the flexibility of PacSolve by allowing these different components to be flexibly enabled, disabled, and configured. We then conduct an extensive empirical evalution by using Repyro to perform dependency repair on datasets of Python Gists [46] and Python Jupyter notebooks [79], and perform an ablation study to understand what effects the components of Repyro have on dependency repair, both separately and when working in concert.

Our results suggest promising directions in the field of automated dependency repair. Specifically, we show that by combining multiple techiques together we can successfully repair more programs while doing so in fewer search iterations. Repyro also offers key qualitative features compared to prior work. First, Repyro returns mutated dependency *constraints* as opposed to dependency versions, which provides programmers a higher-level understanding of what the underlying repairs are, and allows them to integrate

with other systems more easily. Second, REPYRO (through its use of LLMs) is able to reason about and report when a program is failing for non-dependency reasons, and thus a *partial* repair has been found. Our results suggest that many programs fail due to non-dependency causes as well, so enabling dependency repair tools to reason about this and communicate to the user is a crucial step towards making dependency repair practical.

6.1 THE DEPENDENCY REPAIR PROBLEM

To our knowledge, all existing work on automated dependency repair works by a search process of iteratively applying mutations to dependency solutions, based on heuristics derived from error messages or other sources [48, 70]. Starting with a dependency solution (σ_1) derived from the buggy constraints (e.g. by using standard Pip), the program (P) is then executed in the context of dependency solution σ_1 and produces an error:

$$\sigma_1, P \Downarrow \mathrm{Err}(E)$$

where $\sigma_1, P \Downarrow v$ indicates the language-specific operational semantics of evaluating a program in an environment populated with a dependency solution.

Next, some mutation operators are applied to σ_1 to derive a new dependency solution σ_2 to test. Commonly the mutation operators are based on the error message E, but prior work has also suggested mutating based on offline data analysis, such as package version compatibility statistics mined from build logs of open source projects [48]. Dependending on the exact search algorithm, multiple new dependency solutions may be derived, and prioritized heuristically. Overall, this approach of mutating dependency solutions presents a path for representing the problem of dependency repair as a fairly standard graph search problem, to which creative search heuristics can be applied.

Formally, these repair algorithms are given a program (P) and attempt to find a dependency solution (σ') which causes the program P to run successfully:

$$\sigma', P \Downarrow ok$$

However, we argue that this is not the most useful formulation of the problem. Programmers care not only to find a one-off solution that recovers the functionality of the program, but also to determine how to fix the buggy dependency constraints while retaining constraint flexibility. For example, a developer of a package may have written dependency constraints that are too broad, and their package no longer

functions correctly when solved with up-to-date dependencies. If the developer obtains a dependency solution σ' which causes their package to work, this does not help them know how to correctly restrict their dependency constraints, short of fully pinning all dependencies which may be undesirable to clients.

Instead, we consider dependency repair to be providing programmers with repaired constraints, which when solved with a dependency solver yield a dependency solution under which the program runs successfully. Suppose that $\mathcal{S}$ is a dependency solver. We say that dependency constraints Σ' repair the program P for solver $\mathcal{S}$ if:

$$\mathcal{S}(\Sigma'), P \Downarrow \mathit{ok}$$

Once a programmer obtains dependency constraints that repair the program, then they can simply use those constraints going forward, or compare them to the original constraints to help understand the root of the problem.

6.2 THE REPYRO ARCHITECTURE

To achieve repair at the level of dependency constraints, we propose performing search mutations on dependency constraints rather than dependency solutions (versions), and selecting these mutations based on obtaining concrete dependency solutions and executing them. This approach has several promising properties. First, if we perform mutations on constraints themselves, then the suggested aggregate constraint-level repairs are readily available from the mutation history. Second, mutating constraints allows us to cutout potentially massive sections of the dependency solution space very quickly, which may aid in finding repairs more efficiently. Finally, different types of heuristics (LLMs and PACSOLVE optimization objectives) can be given separate but related roles (constraint mutation and constraint solving), leading to a natural method for integrating LLMs with traditional cost function directed search.

Fig. 6.1 illustrates the main components of REPYRO, our tool which performs dependency repair by iteratively mutating dependency constraints. REPYRO starts with the initial (buggy) dependency constraints, Σ. REPYRO then iteratively interleaves dependency solving, program execution, and constraint mutation as follows. First, REPYRO invokes the dependency solver (we use PACSOLVE but any other solver can work) to find a dependency solution σ_1. Then, the program P is executed[1] under σ_1, and produces an error. Next, constraint mutation occurs, in which some constraint mutation tactic

1 or tests of P may be executed, if REPYRO is configured to do so.

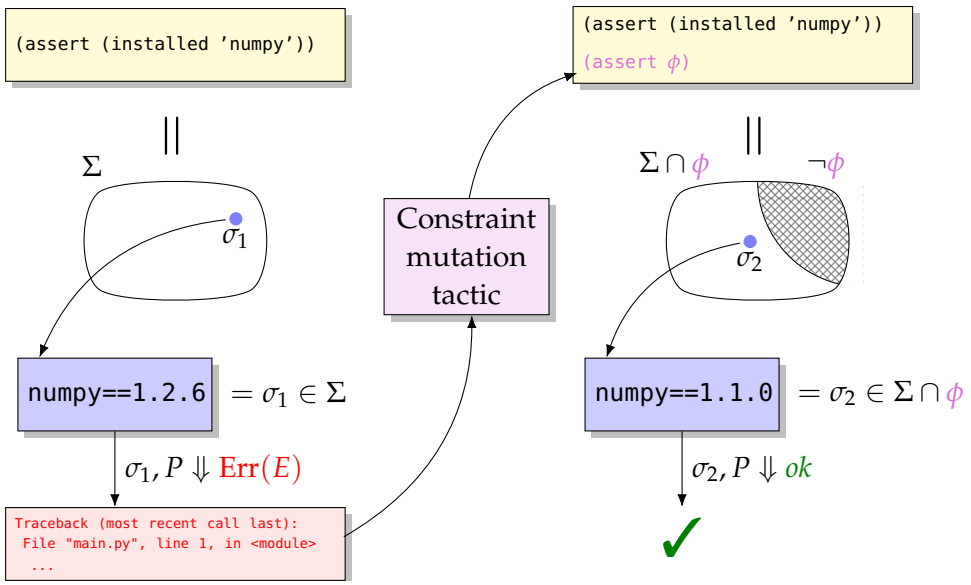


Figure 6.1: REPYRO: Dependency repair by interleaved constraint solving and constraint mutation

mutates the constraints Σ. Various forms of constraint mutation are possible, but in this work we restrict ourselves to mutations which only conjoin new constraints with existing constraints, thus obtaining constraints that are more restrictive than before ($\Sigma \cap \phi$). Now on the next iteration, the dependency solution space is significantly reduced, as all dependency solutions in $\neg\phi$ are now eliminated (grayed-out zone). The dependency solver thus chooses a new solution σ_2 within the constrained space. This process is repeated until a solution is found for which the program executes successfully.

The REPYRO framework is quite general. In particular, it subsumes the approach of performing mutations on dependency solutions rather than constraints, as dependency solutions can simply be represented as fully pinned dependency constraints, and then the constraint mutator and solver only operate on fully pinned dependencies. Of course in this work we instantiate REPYRO with components that enable it to truly perform mutations on flexible constraints. The main building blocks that we need to instantiate REPYRO with are thus: a) the dependency solver, which we use PACSOLVE for, b) the optimization objectives for PACSOLVE, and c) the constraint mutation tactics. By bulding on PACSOLVE, we can use optimization objectives which heuristically prioritize dependency solutions that are more likely to succeed. At each iteration of REPYRO,

we are thus more likely to encounter a working solution and terminate the search. Additionally, the flexibility of PacSolve enables us to easily configure or disable these heuristics, which we leverage in our empirical evaluation (Section 6.3).

6.2.1 *Modeling Python Dependencies with* PacSolve

Repyro is a tool specifically for repairing constraints in the Python ecosystem. To use PacSolve as the underlying dependency solver for Repyro, we first have to encode Pip constraints and semantics into PacSolve. For the most part, Pip uses fairly standard semantic versioning style version numbers and constraints, so the encoding generally follows that of NPM in Chapter 5. However, there are a couple of key differences that we need to handle:

1. Python only supports exactly 1 version per package, so the diagonal consistency function must be used (Section 4.3.2),
2. Pip allows packages to have *optional dependencies*, which are enabled by named feature flags, e.g. `requests[security]`, and
3. Python versions themselves should be a solvable variable, as selecting the right Python version is (anecdotally) important for repair.

To encode optional dependencies gated by feature flags, we encode every feature flag (F) of a package (P) as an auxillary package ($P.F$) with versions that correspond to the versions of the package that have that feature flag. The auxillary package ($P.F$) is then given the optional dependencies that are gated behind that feature flag, along with an exact equality dependency on the correspoding version of the package (P), which ensures that selected feature flag versions are kept in sync with the main package versions. Then, any dependency constraint that contains a feature flag, e.g. `P[F] >= 1.2.3`, is encoded into a dependency on the package ($P >= 1.2.3$), and an unrestricted dependency on the auxillary package ($P.F$).

Enabling Python versions to be solvable by PacSolve was more challenging however, because Pip allows dependencies to be selectively enabled based on Python versions. Implementing this required extending PacSolve with support for conditional dependency constraints, that is, dependencies which are selectively enabled based on some boolean constraint involving possibly other packages (in this case, a distinguished `python` package).

6.2.2 *Optimization Objectives for Dependency Repair*

When using the PACSOLVE framework to build a dependency solver, we also need to configure the optimization objective. Chapter 5 explored several optimization objectives, including `minGoal-oldness` which minimized the sum of the ranks of dependency versions when sorted by release date. In REPYRO we do not want to prefer the newest dependency solutions, as those solutions are typically what is already being produced by the buggy constraints when solving dependencies with Pip. Instead of minimizing oldness, we minimize time difference between package versions and some known date (T_0) at which the program in question (presumably) functioned correctly using dependencies solved by Pip. Such a known date is typically mined from sources such as Git commit dates, file metadata, or other similar sources. Note that there is no guarantee that the program did in fact work at this point in time, as it may have been buggy then as well, but it reflects the best knowledge we have about a point in time in which the programs may have worked. We define two different optimization objectives for finding solutions that contain package versions that are near this known date:

$$\text{Days}_{T_0}(\sigma) = \sum_{d \in \text{deps}(\sigma)} |T_0 - \text{date}(d)|$$

$$\text{Rank}_{T_0}(\sigma) = \sum_{d \in \text{deps}(\sigma)} \left|\left\{d' \mid d' \in \text{versions}(d), \left|T_0 - \text{date}(d')\right| < |T_0 - \text{date}(d)|\right\}\right|$$

The first optimization objective, $\text{Days}_{T_0}(\sigma)$, simply minimizes the total date difference between all dependencies in the solution ($d \in \text{deps}(\sigma)$) and the assumed working date (T_0). The second optimization objective, $\text{Rank}_{T_0}(\sigma)$, follows the structure of `minGoal-oldness` in Chapter 5, minimizing the rank of a dependency d among other versions of the same package when sorted by time away from the assumed working date. Note that because REPYRO includes the Python version as a solvable variable, the release date of the selected Python version will be among the terms of these optimization objectives. Since both objectives are linear over the dependencies in the solution, they are easy to express and implement in the PACSOLVE framework.

6.2.3 *Mutation Tactics and Integrating LLMs*

The last building block for REPYRO is the *constraint mutation tactic*. REPYRO provides a flexible abstraction allowing arbitrary tactics to be implemented, two core pre-built tactics (discussed shortly), and tactic combinators such as sequencing (apply two tactics).

Repyro can then be configured by specifying which tactic (including those derived through combinators) should be used for constraint mutation.

While Repyro's tactic framework allows tactics to perform arbitrary mutations to constraints, based on error messages, current solutions, and current constraints, for this work we focused on restricted tactic forms, which only consider the program's error message (E) and current solution (σ_i), and produce constraints which are conjoined with the current constraints (Σ_i):

$$\Sigma_{i+1} = \Sigma_i \wedge f(E, \sigma_i)$$

This restricted form allows Repyro to perform an important solving optimization. Because the next constraints always represent a subset of solutions from the current constraints ($\Sigma_{i+1} \subseteq \Sigma_i$), PacSolve does not need to redo its expensive solution graph sketching algorithm, which includes fetching from PyPI all potentially useful packages and versions. Instead, the same solution graph can be reused, with additional constraints on it.

CURRENT SOLUTION EXCLUSION TACTIC The first core tactic that Repyro provides is extremely simple. The *excludeCurrent* tactic lifts the current solution into a constraint by conjoining exact equality constraints for each dependency in the solution, and negates this current solution constraint:

$$\text{excludeCurrent}(\sigma_i) = \neg\sigma_i$$

While simple, *excludeCurrent* is an important component, as it reflects the obvious fact that we should never retry a solution that already failed, and enables Repyro to be used in simple configurations for performing repair without the use of LLMs or more advanced tactics. We leverage this to great extend in our evaluations (Section 6.3).

LARGE-LANGUAGE MODEL TACTIC The other core tactic, *promptLLM*, is a tactic which prompts an off-the-shelf large-language model with the current error message (E) along with instructions to produce a new PacSolve constraint that would resolve the error message. Additionally, we prompt the LLM to produce a special **GIVEUP** token if the given error can not be fixed through Python-level dependencies. This includes both non-dependency bugs (e.g. failed to load CSV files, etc.) and system-level dependency errors (missing `libcurl`, etc), as they are not representable by Repyro and we make no attempt to repair them. We use few-shot prompting to show 2 examples of

error messages and example constraint fixes written in an adapted syntactic form of the PACSOLVE DSL, and 1 example of a non-dependency error and the **GIVEUP** token. The full prompt template can be found in Appendix A.

Whenever we use the *promptLLM* tactic in REPYRO, we always pair it with the *excludeCurrent* tactic via the sequencing combinator, that is, we define:

$$\text{LLM} := \text{Seq}(\text{excludeCurrent}, \text{promptLLM})$$

Throughout the evaluation section (Section 6.3) we will always use this variant of LLM-directed repair.

Other LLM-based techniques could certainly be explored here, including fine-tuning LLMs on this task, and/or performing searches for similar error messages on Stack Overflow and using those answers as context in the prompt (retrieval-augmented generation [58]). In this work we focus our contributions on developing the REPYRO framework (which can easily be reused in future work to try more advanced techniques) and evaluating how well off-the-shelf LLMs perform with minimal prompt engineering.

6.3 EVALUATION

As discussed above, the general framework of REPYRO constructs a family of concrete automated dependency repair tools by allowing multiple heuristics and tactics to be combined in how they guide the search process. In this empirical evaluation we aim to understand what impact these indiviual configurable components have on the ability to repair dependencies. Specifically, we answer the following five research questions:

- **RQ1:** How many Python projects become unrunnable over time due to dependency errors?
- **RQ2:** Can dependency repairs be found through undirected search?
- **RQ3:** Do date-based heuristics aid REPYRO's search by finding more successful solutions and reducing the number of search steps?
- **RQ4:** Can off-the-shelf LLMs aid REPYRO's search by finding more successful solutions and reducing the number of search steps?
- **RQ5:** Are these techniques complementary to each other in aiding repair?

Additionally, we look at various examples of successful and failed fixes relevant to each technique to qualitatively understand their relative strengths.

6.3.1 *Datasets*

Evaluation of REPYRO is performed over two distinct types of Python programs: Python Gists and Python Jupyter notebooks. Table 6.1 describes the relevant statistics for each dataset we evaluate on. These datasets are described in detail below.

Dataset	N	Mean #deps	Median #deps	Min #deps	Max #deps
Gistable$_1$	100	1	1	1	1
Gistable$_2$	100	2	2	2	2
Gistable$_{\geq 3}$	217	3.4	3	3	8
Julynter	69	3.9	4	1	11

Table 6.1: Sizes of the datasets, and statistics of number of direct dependencies in each dataset

6.3.1.1 *Gistable*

The dataset of Gists is derived from the Gistable dataset [46]. In [46] the authors mined a large dataset of Python Gists from GitHub. They then performed a naive form of dependency inference to infer which PyPI packages to install in order to get Python Gist to execute successfully, by installing PyPI packages with the same names as import statements in the Gist source code, without regard for versions of packages. With this naive approach, the authors were able to get 4,945 Gists to execute successfully.

Each Gist which had dependencies inferred successfully in the Gistable dataset comes with a Dockerfile which lists the inferred and unpinned dependencies. Because the authors of [46] did not pin the inferred dependencies, many of these Gists are now broken due to dependency drift. We thus take the unpinned inferred dependencies as the constraints which we wish to test as possibly buggy, and if so repair using REPYRO. To make our evaluation more tractable, we also subset the Gistable dataset, while oversampling the Gists with more dependencies, so that we have sufficient numbers of the more complex and difficult Gists to repair. We uniformly subsample 100 Gists that have exactly 1 dependency (*Gistable*$_1$), again uniformly subsample 100 Gists that have exactly 2 dependencies (*Gistable*$_2$), and take all 217 Gists that have 3 or more dependencies (*Gistable*$_{\geq 3}$). Through this segmentation we ensure that we have sufficient quantities of Gists across the distribution of number of dependencies, and can observe how REPYRO performs on these datasets of varying challenge.

Dataset	Dep Err	Other Err	Timeout	Success
Gistable$_1$	13% (13)	49% (49)	4% (4)	34% (34)
Gistable$_2$	33% (33)	36% (36)	9% (9)	22% (22)
Gistable$_{\geq 3}$	51% (110)	35% (75)	6% (12)	8% (18)
Julynter	26% (16)	64% (39)	2% (1)	8% (5)
Total	**36% (172)**	**42% (199)**	**5% (26)**	**17% (79)**

Table 6.2: Program execution results when installing dependencies with Pip

6.3.1.2 *Julynter*

The dataset of Jupyter notebooks is derived from the Julynter dataset [79]. The authors of [79] randomly selected a representative subsample of 69 of their mined Jupyter notebooks. We use this as our set of Jupyter notebooks to evaluate on. The notebooks did not necessarily come with any inferred dependencies, but rather with a clone of the entire GitHub repository they were mined from, which would sometimes contain `requirements.txt` or other similar files listing dependencies. For any notebooks for which there were no listed dependencies, we manually analyzed all the import statements and added the relevant PyPI package to the list of dependencies. After this process, we had a list of unpinned dependencies which we take to be the initial, possibly buggy constraints to repair. We did not segment the *Julynter* dataset so as to avoid creating subgroups with too few members.

We also retain the clone of each notebook's GitHub repository. When Repyro executes notebooks, we are careful with our execution infrastructure to run the notebook with an appropriate working directory to allow the notebook to load data files or other Python modules contained in its repository.

6.3.2 *RQ1: How many Python projects become unrunnable over time due to dependency errors?*

First, we aim to estimate the frequency of dependency errors on Python projects within our datasets, both to understand the impact of dependency drift on the reproducibility of software and to set baselines for our repair techniques.

One challenge with this is that both Python 2 and Python 3 are used by programs in the datasets, and we do not know a priori which version of Python was used by each program. While this is itself an instance of dependency drift, if we were to simply only run with one version of Python (e.g. Python 3), we would observe such a large quantity of failures due only to this issue that we would effectively mask all the other types of dependency and non-dependency errors.

Instead, we give each program the best shot possibly by running it under both Python 2.7.18 and Python 3.12.2 using the exact same containerized infrastructure used by REPYRO. The (unpinned) dependencies for each program are installed using standard Pip. We then examined the result of executing each program under both Python versions, and manually classified the root cause of each error message as being caused by a dependency error or not. Errors that were due to the wrong Python version or missing or wrong versions of packages were counted as a dependency bug root cause. We then combined the Python 2 and Python 3 results by only counting a program as having a dependency related error if it had a dependency related error under both versions of Python. For example, if a program P crashes due to the wrong Python version when run under Python 2, and crashes due to a missing dependency when run under Python 3, then P would be categorized as having a dependency root cause. But if P where to crash due to failing to load a CSV file when run under Python 3, then P would be categorized as having a non-dependency root cause.

Note that a program which exhibits a non-dependency related crash at runtime may in fact also contain a dependency related bug(s) which was not reached at runtime, and thus the numbers of dependency errors found in this analysis should be considered as a lower bound. Additionally, some of the non-dependency errors are the result of the specific instructure used in our experimental setup; in particular several gists and notebooks required access to a writable filesystem, while our infrastructure provided a read-only filesystem.

Table 6.2 shows the frequency of execution results in each of the datasets, considering the results of running with both Python 2 and 3, and using Pip to install the (unpinned) dependencies. Overall, we see that only 17% of all tested programs run successfully (exited with status code 0). The error types are split fairly evenly between dependency related errors (36%) and other errors (42%). A small fraction of programs timed out after 40 minutes. This indicates that while other issues beyond dependency problems are clearly important for software reproducibility, resolving dependency problems is a required ingredient for improving reproducibility.

When comparing the Gists datsets, we see that Gists with more dependencies have higher rates of dependency errors ($Gistable_1$ = 13%, $Gistable_2$ = 33%, and $Gistable_{\geq 3}$ = 51%), and lower rates of non-dependency errors. Jupyter notesbooks appear however to be distributionally quite different from Gists, as a significantly higher percentage of them fail due to non-dependency errors. This in line with the findings from [79], which show that Jupyter notebooks are frequently prone to many notebook-specific reproducibility bugs (out of order cells, missing CSV files, etc.), in addition to reproducibility issues faced by standalone small scripts (Gists). Note that in our experimental setup we took great care to ensure that notebooks are run as correctly in their environment as possible, including copying all data and additional Python files into appropriate directories so they could be loaded by the notebook. We did not attempt to run notebook cells out-of-order.

> **RQ1:** 36% of Python programs in our dataset immediately present with dependency errors upon execution, with the percentage increasing for programs with more dependencies

6.3.3 *RQ2: Can dependency repairs be found through undirected search?*

The simplest configuration of REPYRO performs undirected search of the dependency solution space. Before evaluating the guided search procedures available in REPYRO, we first examine how well this naive search procedure performs in practice. We run REPYRO configured to perform undirected search, and time it out after 40 minutes. This search procedure thus produces exactly three possible outcomes: repaired (a solution under which the program succeeds is found), failure (the entire PACSOLVE search space is exhausted), or timeout (after 40 minutes in total). Note that a timeout may be due to either the Python program itself taking too long to execute, or too many iterations of search occurring. We additionally track how many iterations of search were performed.

Table 6.3 displays the frequency of the 3 possible outcomes when running unguided REPYRO on the Gists and notebooks which failed due to dependency errors as reported in Table 6.2. We do not run REPYRO on programs that initially had other types of errors or succeeded because we are attempting to understand how effective REPYRO can be at performing repairs, not at general dependency solving. Additionally, this reduces the effects of flaky programs on our analysis. The table shows that more dependencies leads to a lower success rate and higher timeout and failure rates.

Dataset	Repaired	Timeout	Failure
Gistable$_1$	23% (3)	69% (9)	8% (1)
Gistable$_2$	15% (5)	33% (11)	52% (17)
Gistable$_{\geq 3}$	4% (4)	40% (44)	56% (61)
Julynter	0% (0)	94% (15)	6% (1)
Total	**7% (12)**	**46% (79)**	**47% (80)**

Table 6.3: Results when running unguided repair search

Second, the table shows that Repyro fails in a large number of cases (47% overall). The vast majority are due to dependency constraints being unsatisfiable at the start, that is, on the very first iteration PacSolve immediately responds with UNSAT. Manually inspecting these cases reveals that the underlying cause is that packages or versions of packages which are required have been removed from PyPI. While outside the scope of this work, a fairly straightforward extension to Repyro could configure PacSolve to treat dependency constraints as soft constraints, so as to allow Repyro to proceed with searching for repairs in these cases.

An example of a successful repair with unguided Repyro is Gist `6049407`, which depends on packages which require Python 2. For several iterations Repyro attempts various Python 3 solutions, all of which fail. On the 9th iteration Repyro happens to switch to selecting Python 2, which in this case is sufficient to succeed.

Fig. 6.2 shows the cumulative number of successful repairs found per iteration, across multiple configurations of Repyro. While the other configurations of Repyro will be discussed in following RQs, at the moment we can read the pink (Unguided) line to understand how unguided search behaves over time. Unguided search initially finds 7 successful repairs, however must then take many iterations (close to 30) to find all repairs it is able to (within the 40 minute timeout).

RQ2: Repyro can indeed find some dependency repairs through entirely undirected search. However, it takes many search iterations (close to 30) to do so in some cases.

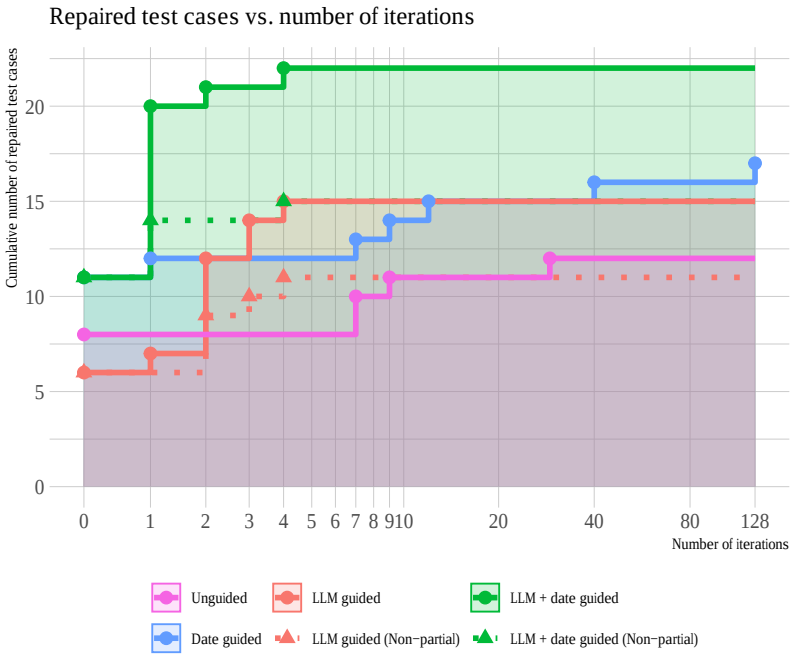


Figure 6.2: Cumulative number of successful repairs at each iteration count. Each line represents a configuraiton of REPYRO, and a point at (x, y) indicates that the configuration was able to repair y programs in x or fewer iterations. The x-axis is pseudo-log scaled. Dotted lines indicate non-partial success metrics for LLM-based models.

6.3.4 *RQ3: Do date-based heuristics aid* REPYRO*'s search by finding more successful solutions and reducing the number of search steps?*

REPYRO inherits PACSOLVE's flexibility for configuring optimization objectives. We use this to configure REPYRO with optimization objectives and optionally additional constraints that guide the search process. Specifically, every program in our dataset has an estimated date at which it presumably worked correctly (T_0): Gist dates are determined by the date the Gist was published to GitHub, and Jupyter notebook dates are determined by the date of the Git commit from which they were mined by the authors of Julynter.

Based on this mined date, we evaluate applying either of the two optimization objectives discussed in Section 6.2.2: $\text{Days}_{T_0}(\sigma)$ and $\text{Rank}_{T_0}(\sigma)$. Additionally, we define optional constraints that restrict the search space based on T_0 to only select dependency versions which existed at or before T_0. In total, these options yield four different repair configurations: $\text{REPYRO}_{\text{Days}}$, $\text{REPYRO}_{\text{Rank}}$, $\text{REPYRO}^{\leq T_0}_{\text{Days}}$, $\text{REPYRO}^{\leq T_0}_{\text{Rank}}$. Across these four configurations, we repeat the experiment from Section 6.3.3, and observe how the date-based heuristics affect the number of repairs and the search dynamics.

Table 6.4 shows the final results of running all four configurations of date-based heuristics. As with the unguided search, this variant of REPYRO produces exactly three possible outcomes: repaired, timeout, or failure. Additionally, we indicate with ($+x$ / $-y$) the number of *paired* gained and lost programs in each category compared to unguided search. Overall, the configurations which order the search space by date heuristics ($\text{REPYRO}_{\text{Days}}$, $\text{REPYRO}_{\text{Rank}}$) find more fixes than the unguided search, finding 3 and 5 more net fixes in total each. Moreover, the gains in performance come entirely from turning cases of timeouts in successful repairs. However, as the date guidance is a *heuristic*, there are a few cases where unguided search found a repair while date guided search did not within the timeout. As expected, there are no effects on the number of failures for the unconstrained.

An example of successful date heuristic guided repair is Gist `11005158`, which imports the `JSONEncoder` class from the `flask.json` module. However, version 2.3.0 of Flask removed the `JSONEncoder` class. Based on the date of the Gist, REPYRO selects an older version of Flask (0.10.1) which causes a success on the first try.

The constrained configurations ($\text{REPYRO}^{\leq T_0}_{\text{Days}}$, $\text{REPYRO}^{\leq T_0}_{\text{Rank}}$) find fewer successful fixes and substantially higher failure rates. This indicates that the date guidance can only be used as a heuristic and not as a hard constraint, as it is sometimes necessary for repair to consider package versions that did not yet exist when the program was committed. One hypotheses for this effect is that programmers may have uploaded Gists which highlight a bug in a current version of a package.

Returning to Fig. 6.2, the blue line (Date guided) plots the cumulative number of successful repairs found per iteration for $\text{REPYRO}_{\text{Rank}}$, the best performing date-guided configuration. Comparing to the unguided configuration, the date guidance helps REPYRO immediately find more fixes in the initial iteration. After that, the date-guided search then behaves similarly, finding a handful of additional fixes around 10 iterations, and a few more after a large number of iterations (40 and 128).

RQ3: Date-based heuristics help REPYRO to find more successful repairs in fewer iterations. However, date-guided search (like unguided search) has a long tail of the number of search iterations. Additionally, constraining by date cuts out too much of the solution space.

Config	Dataset	Repaired	Timeout	Failure
REPYRO$_{\text{Days}}$	*Gistable*$_1$	23% (3, +0 / -0)	69% (9, +0 / -0)	8% (1, +0 / -0)
REPYRO$_{\text{Days}}$	*Gistable*$_2$	12% (4, +0 / -1)	36% (12, +1 / -0)	52% (17, +0 / -0)
REPYRO$_{\text{Days}}$	*Gistable*$_{\geq 3}$	8% (8, +4 / -0)	35% (37, +0 / -4)	57% (60, +0 / -0)
REPYRO$_{\text{Days}}$	*Julynter*	0% (0, +0 / -0)	94% (15, +0 / -0)	6% (1, +0 / -0)
Repyro$_{\text{Days}}$	**Total**	**9% (15, +4 / -1)**	**44% (73, +1 / -4)**	**47% (79, +0 / -0)**
REPYRO$_{\text{Rank}}$	*Gistable*$_1$	23% (3, +0 / -0)	69% (9, +0 / -0)	8% (1, +0 / -0)
REPYRO$_{\text{Rank}}$	*Gistable*$_2$	15% (5, +1 / -1)	33% (11, +1 / -1)	52% (17, +0 / -0)
REPYRO$_{\text{Rank}}$	*Gistable*$_{\geq 3}$	9% (9, +5 / -0)	32% (33, +0 / -5)	59% (60, +0 / -0)
REPYRO$_{\text{Rank}}$	*Julynter*	0% (0, +0 / -0)	94% (15, +0 / -0)	6% (1, +0 / -0)
Repyro$_{\text{Rank}}$	**Total**	**10% (17, +6 / -1)**	**41% (68, +1 / -6)**	**48% (79, +0 / -0)**
REPYRO$^{\leq T_0}_{\text{Days}}$	*Gistable*$_1$	15% (2, +0 / -1)	15% (2, +0 / -7)	69% (9, +8 / -0)
REPYRO$^{\leq T_0}_{\text{Days}}$	*Gistable*$_2$	9% (3, +0 / -2)	12% (4, +1 / -8)	79% (26, +9 / -0)
REPYRO$^{\leq T_0}_{\text{Days}}$	*Gistable*$_{\geq 3}$	5% (5, +3 / -1)	16% (17, +0 / -25)	79% (84, +23 / -0)
REPYRO$^{\leq T_0}_{\text{Days}}$	*Julynter*	0% (0, +0 / -0)	81% (13, +0 / -2)	19% (3, +2 / -0)
Repyro$^{\leq T_0}_{\text{Days}}$	**Total**	**6% (10, +3 / -4)**	**21% (36, +1 / -42)**	**73% (122, +42 / -0)**
REPYRO$^{\leq T_0}_{\text{Rank}}$	*Gistable*$_1$	15% (2, +0 / -1)	15% (2, +0 / -7)	69% (9, +8 / -0)
REPYRO$^{\leq T_0}_{\text{Rank}}$	*Gistable*$_2$	9% (3, +0 / -2)	12% (4, +1 / -8)	79% (26, +9 / -0)
REPYRO$^{\leq T_0}_{\text{Rank}}$	*Gistable*$_{\geq 3}$	4% (4, +2 / -1)	17% (18, +0 / -24)	79% (84, +23 / -0)
REPYRO$^{\leq T_0}_{\text{Rank}}$	*Julynter*	0% (0, +0 / -0)	81% (13, +0 / -2)	19% (3, +2 / -0)
Repyro$^{\leq T_0}_{\text{Rank}}$	**Total**	**5% (9, +2 / -4)**	**22% (37, +1 / -41)**	**73% (122, +42 / -0)**

Table 6.4: Results when configuring REPYRO with date-based heuristics. Paired additions and loses in each category are reported relative to unguided search, and are indicated by (+x / -y).

6.3.5 *RQ4: Can off-the-shelf LLMs aid* REPYRO*'s search by finding more successful solutions and reducing the number of search steps?*

Config	Dataset	Repaired	Partial repair	Incorrect give up	Timeout	Failure
Llama 3.1 + backtrack	*Gistable*$_1$	23% (3, +0 / -0)	15% (2)	15% (2)	8% (1, +0 / -8)	38% (5, +5 / -1)
Llama 3.1 + backtrack	*Gistable*$_2$	15% (5, +0 / -0)	6% (2)	0% (0)	18% (6, +0 / -5)	61% (20, +3 / -0)
Llama 3.1 + backtrack	*Gistable*$_{\geq 3}$	4% (4, +1 / -1)	0% (0)	0% (0)	32% (35, +1 / -9)	64% (69, +9 / -1)
Llama 3.1 + backtrack	*Julynter*	0% (0, +0 / -0)	6% (1)	19% (3)	69% (11, +0 / -4)	6% (1, +0 / -0)
Llama 3.1 + backtrack	**Total**	**7% (12, +1 / -1)**	**3% (5)**	**3% (5)**	**31% (53, +1 / -26)**	**56% (95, +17 / -2)**
Llama 3 + backtrack	*Gistable*$_1$	15% (2, +0 / -1)	0% (0)	8% (1)	38% (5, +1 / -5)	38% (5, +5 / -1)
Llama 3 + backtrack	*Gistable*$_2$	16% (5, +0 / -0)	3% (1)	0% (0)	16% (5, +0 / -5)	66% (21, +4 / -0)
Llama 3 + backtrack	*Gistable*$_{\geq 3}$	4% (4, +2 / -2)	3% (3)	2% (2)	27% (28, +1 / -13)	65% (68, +8 / -1)
Llama 3 + backtrack	*Julynter*	0% (0, +0 / -0)	0% (0)	0% (0)	87% (13, +0 / -1)	13% (2, +1 / -0)
Llama 3 + backtrack	**Total**	**7% (11, +2 / -3)**	**2% (4)**	**2% (3)**	**31% (51, +2 / -24)**	**58% (96, +18 / -2)**

Table 6.5: Results when configuring REPYRO with LLM-based repair. Paired additions and loses in each category are reported relative to unguided search, and are indicated by (+x / -y).

Next, we evaluate whether off-the-shelf LLMs can help REPYRO find more successful repairs, and how it affects the search dynamics. We evaluate this separately from the date-based heuristics before combining them together (Section 6.3.6). LLM-directed repair in REPYRO works by prompting the LLM at each search iteration to produce additional PACSOLVE constraints which are conjoined with existing constraints.

When configured with LLM-directed repair, REPYRO produces 4 possible outcomes: it may find a repair, timeout, end in failure, or, the LLM may classify the current error as a non-dependency error and give up. Giving up indicates that the program is (as judged by the LLM) partially repaired such that dependency bugs are no longer the cause of the error, and that the programmer should continue with other debugging and remediation steps. When the LLM gives up, we then manually categorize the error as a dependency error or not (as in Section 6.3.2) to determine the ground-truth. If the model is correct, we call this a partial repair, and if not, an incorrect give up.

First, we observe the frequency of these outcomes when REPYRO is configured to use LLM-directed repair. Specifically, we run REPYRO in two configurations: using Llama 3 (70B parameters) and Llama 3.1 (70B parameters). Table 6.5 shows the results of running these two REPYRO configurations across all 4 datasets. As with date-guided search, we show additions and loses in each category relative to unguided search with ($+x$ / $-y$), respectively. Since the give up categories did not exist for unguided search, they do not have relative gains or losses.

Compared to unguided search, LLM-directed repair encounters fewer timeouts, and those are shifted into the give up and failure outcomes. The increase in failures is because the LLM is conjoining additional constraints that do not otherwise exist. Whereas unguided search would only produce a failure if the entire search space was exhausted, LLM-directed repair is continually cutting out entire regions of the search space by adding new constraints. So whether a solution truly is impossible or the LLM erroneously cutout a useful part of the search space, the search procedure arrives at a failure outcome faster than with unguided search. While the failure rate with LLM-synthesized constraints is higher than with unguided search, it is also lower than with date-based hard constraints (Table 6.4). This suggests that the LLM is able to make use of the context of the error message to not erroneously remove as much of the search space, compared to the date-based constraint.

Now comparing the number of correct vs. incorrect give up occurrences, we see that the precision of the model certainly leaves a lot to be desired. While the sample sizes are too small to determine a meaningful false positive rate, it does not appear that if the model classifies an error as a non-dependency error, this fact can be trusted by the programmer. Clearly there are limits to the knowledge of off-the-shelf models with regards to understanding highly specific error messages. Likely this could be improved with more examples in the prompt, offline supervised fine-tuning, RAG-based techniques [58], or interaction with the programmer.

Moving to the successful repairs, we observe a similar number of full repairs as unguided search, but when combining the full repairs with partial repairs, we do see an improvement (+3 net for Llama 3).

An example of a repair synthesized by an LLM is on Gist `5f52ceb565264b1e969a`, which uses the `bs4` (Beautiful Soup) package to scrape video information from YouTube. However, it crashes with an error message of "Malformed attribute selector". This is because version 4.7.0 of `bs4` made selector syntax more strict (compliant with CSS), causing some previously allowed selectors to raise an exception [7]. In response to this error message, the LLM elects to constrain the version of `bs4` to be less than version

4.11, which is not actually a sufficient fix. While the LLM is able to determine which package is faulty based on the error message, it is not able to determine the correct version to rollback to. However, in this case the LLM's constraint does not exclude the repairing solution from the space, and indeed Repyro then produces a solution with a sufficiently old version of `bs4`. This example highlights the capabilities and limitations of off-the-shelf LLM-directed repair.

When considering the search dynamics in Fig. 6.2 again, we see that the repairs (both partial and full) occurr much sooner with LLM-guided search. The solid orange line (LLM guided) shows the cumulative combined full and partial repairs found with LLM guided search per iteration, while the dashed line shows full repairs only. In both metrics, the LLM guided search performs multiple repairs quite early (within 4 iterations), and at that point has improved upon unguided search. As unguided search proceeds, it eventuallly (around 30 iterations) overtakes LLM guided search when considering full repairs only. In fact, all LLM-based repairs occurr within the first 4 iterations.

When comparing how LLM guidance and date heuristic guidance affects the search dynamics, we see that the two methods affected the search quite differently. The date-based heuristics made some problems trivially solvable in 0 steps, but other problems remained in the long tail which would take many iterations to solve. On the other hand, LLM-synthesized constraints primarily work to eliminate the long tail of extremely long search processes. Thus, the LLMs appear quite effective at limiting the size of the search space. One hypothesis for the LLMs' middling number of successful repairs in Table 6.5 is that because date-based heuristics are not used, the LLMs are being asked to respond to error messages from extraodinarily ancient and likely irrelevant versions of packages, and in doing so they may produce constraints which effectively reduce the search space but also cut out useful solutions.

While the LLM by itself does not improve full repairs compared to unguided search, it does add a crucial capability of allowing the search to return a partial repair, which is important in practice, and allows some repairs to happen substantially sooner.

> **RQ4:** LLM-synthesized constraints effectively help Repyro to limit the search space, and find roughly the same number of successful repairs in fewer iterations. LLMs are not (by themselves) able to substantially improve the rate of repairs.

6.3.6 *RQ5: Are these techniques complementary to each other in aiding repair?*

The results explored in Section 6.3.4 and Section 6.3.5 suggest that date-based heuristics and LLMs may help the dependency repair search process in different ways: date-based heuristics help to provide more relevant concrete solutions at any given iteration, while LLMs provide the ability to effectively reduce the search space and give up so that (partial) results can be returned to the programmer in a reasonable amount of time.

We now configure REPYRO to use both date-based heuristics (REPYRO$_{\text{Rank}}$) and LLM-guided repair (Llama 3). At each iteration, REPYRO finds the best dependency solution according to the date-based heuristics, runs it, prompts the LLM with the resulting error message, and conjoins the LLM's synthesized PACSOLVE constraint with the current constraint set. As in Section 6.3.5, REPYRO can conclude with repaired, partially repaired, timeout, failure or incorrect give up.

Config	Dataset	Repaired	Partial repair	Incorrect give up	Timeout	Failure
REPYRO$_{\text{Rank}}$ + Llama 3	*Gistable*$_1$	25% (3, +0 / -0)	0% (0)	8% (1)	42% (5, +1 / -4)	25% (3, +3 / -1)
REPYRO$_{\text{Rank}}$ + Llama 3	*Gistable*$_2$	16% (5, +1 / -1)	9% (3)	0% (0)	9% (3, +0 / -7)	66% (21, +4 / -0)
REPYRO$_{\text{Rank}}$ + Llama 3	*Gistable*$_{\geq 3}$	7% (7, +4 / -0)	5% (5)	3% (3)	23% (24, +0 / -15)	62% (64, +3 / -0)
REPYRO$_{\text{Rank}}$ + Llama 3	*Julynter*	0% (0, +0 / -0)	0% (0)	0% (0)	73% (8, +0 / -2)	27% (3, +2 / -0)
Repyro$_{\text{Rank}}$ + Llama 3	**Total**	**9% (15, +5 / -1)**	**5% (8)**	**3% (4)**	**25% (40, +1 / -28)**	**58% (91, +12 / -1)**

Table 6.6: Results when configuring REPYRO with date-based heuristics and LLM-guided repair. Paired additions and loses in each category are reported relative to unguided search, and are indicated by (+x / -y).

Table 6.6 shows the outputs of running this configuration of REPYRO across all the datasets. As before, we note the additional and lost programs within each category compared to unguided search. Overall, we see that the combination of LLM and date-guided search offers improvements over the baseline of undirected search, with a net 4 additional full repairs and a net 8 additional partial repairs. Ultimately, programmers care that a tool solves their immediate problem of a dependency error, and to that end combining LLM and date guidance yields the best success rate of any tested configuration. Additionally, as with LLM-only search, the timeouts were reduced substantially.

Compared to LLMs alone, the combination of LLMs and date heuristics yields more fixes and more partial fixes. One hypothesis is that due to the date-based heuristics, the LLM is less likely to be prompted with error messages from very old package versions, and doesn't go down the "wrong track" of attempting to fix errors that do not need to be fixed. Compared to date-guidance alone, the combined configuraiton does not find more full repairs, but does remain competetive. This suggests that there is still work to be done on preventing the model from synthesizing constraints which eliminate useful regions of the solution space. When considering partial repairs, the combined configuration substantially out-performs date-only guidance.

However, the difference in performance is starker when considering the number of iterations to repair. Returning to Fig. 6.2, the solid green line (LLM + date guided) shows the cumulative combined full and partial repairs found with the combined configuration per iteration, while the dashed line shows full repairs only. If considering only full repairs, the combined configuration finds the same number of repairs in 4 iterations as date-guided does in over 10 iterations. When looking at partial repairs though, the combined configuration finds substantially more repairs than any other technique, and does so within only 4 iterations. Overall, combining LLMs with date-based heuristics improves not only the number of successful (partial) repairs, but does so in fewer iterations.

An example of a successful repair using both date and LLM guided repair is Gist `4c501fc99acb75852756a4d1dfc8ca3d` which uses Beautiful Soup to scrape sources of free food from Postmates and when found send an SMS alert using the Twilio library. When using Twilio, it imports the `TwilioRestClient` class, but version 6 of Twilio renamed this class to `Client`. In this case, the LLM correctly and precisely responds with a fix of constraining Twilio to a version before 6. By then using date-based optimization, REPYRO completes the repair by producing a solution containing the most up-to-date version of Twilio satisfying that constraint, rather than a likely broken, archaic version. Indeed, the version of Twilio that REPYRO finds that repairs this Gist (5.7.0) is in fact newer than what most Stack Overflow answers suggested rolling back to (5.6.x) [49].

RQ5: Combining PACSOLVE's global optimization objectives with LLM-driven repair yields more successful (partial) repairs than with either technique individually, while doing so in fewer iterations.

6.4 DISCUSSION

The results of our evaluation provide several insights into the effectiveness of REPYRO and its various configurations for automated dependency repair in Python projects.

First, our investigation in RQ1 reveals a significant prevalence of dependency-related errors in Python projects over time, with 36% of programs in our datasets immediately presenting dependency errors upon execution. This underscores the importance of addressing dependency drift and highlights the potential impact of automated repair tools like REPYRO.

The evaluation of undirected search in RQ2 demonstrates that while it can find some dependency repairs, it often requires many iterations to do so, suggesting that more sophisticated search strategies could potentially improve the efficiency of the repair process.

Our exploration of date-based heuristics in RQ3 shows that they can indeed aid REPYRO's search by finding more successful repairs in fewer iterations. In particular, date-based heuristics increase the likelihood of finding a repairing solution on the very first iteration, but has little or no impact after that. This is likely because there are many very similar solutions nearby to the global optimum, so if the global optimum does not repair the dependencies, then search after that will continue sampling highly similar solutions rather than exploring more broadly.

The investigation of LLM-guided repair in RQ4 reveals that off-the-shelf LLMs can effectively limit the search space and find repairs in fewer iterations. While they don't substantially improve the rate of full repairs compared to unguided search, they introduce the crucial capability of allowing partial repairs and earlier termination of the search process. We note that this work focused on building the foundational framework of REPYRO to be able to construct dependency repair tools which integrate LLMs with Max-SMT techniques, and evaluating it with the simplest possible LLM-based approach. Substantially better results may be obtained through more work on the LLM portion of the tool, including fine-tuning models or performing error message search queries and injecting the results into the prompt.

Finally, the combination of date-based heuristics and LLM-guided repair explored in RQ5 demonstrates the complementary nature of these techniques. We hypothsize that the date-guidance helps yield solutions and error messages that are more relevant to the situation at hand, and consequently the LLM can respond to the error message more effectively. In the other direction, the constraints the LLM synthesizes then forces PACSOLVE to find a new global optimum in an area of the search space it had not yet

explored. This combined approach yields more successful (partial) repairs than either technique individually, while also reducing the number of iterations required.

6.5 THREATS TO VALIDITY

6.5.1 *External Validity*

Our usage of the Gistable and Julynter datasets may not be representative of Python programs more broadly. In particular, the Gistable dataset is heavily weighted towards requiring Python 2, likely at a much higher rate than any new Python code being written now. Studying dependency drift and dependency repair requires collecting and using programs which are years out of date, so that you have organic instances of dependency drift to study. Additionally, any other questions of representativeness of the original datasets applies here, and we defer to their work for details and discussion of how the datasets were collected.

In REPYRO we only considered repair for Python programs in the PyPI ecosystem. While we believe the techniques of REPYRO could reasonably translate to other languages and ecosystems (since it is based on PACSOLVE), we do not know how effective the techniques would be in other environments, or if other ecosystems have similar rates of dependency drift to begin with.

6.5.2 *Construct Validity*

In Sections 6.3.5 and 6.3.6 we examined the performance of LLM-based dependency repair. Two metrics, full repairs and partial repairs were considered. While we believe that partial repairs are more indicative of what is useful to the programmer, the selection of the metric construct affects the interpretation of the data. If selecting full repairs as the primary metric, then there is not evidence that LLMs helped find more repairs, only that they helped find repairs in fewer iterations.

6.6 CONCLUSION

REPYRO does not claim to fully solve the problem of automated dependency repair. In contrast, many of the results in Section 6.3 show how much work there is left to do. However, REPYRO offers a promising new avenue for automated dependency

repair, one in which constraints formalized in a language such as PacSolve, heuristics based on solution graph optimization (e.g. date-base optimization), and heuristics for how constraints should be mutated can be composed naturally and seamlessly. Section 6.3 suggests that all of these components are necessary pieces to achieve success in automated dependency repair, and indeed we suspect that quite a bit more juice may be squeezed from this framework, by augmenting any of those three components. For example, one could perform some approximate static analyses to determine which library versions are definitely incompatible with the programmer's code or dependencies' code, and then insert these incompability constraints into PacSolve. Likewise, the problem could be attacked from the LLM direction by exploring, for example, specialized model training techniques. Repyro offers the framework for this exploration, and a substantial portion of its flexibility is due to starting with a solid and flexible foundational formal model in PacSolve.

7

RELATED WORK

7.1 EMPIRICAL ANALYSES OF DEPENDENCY MANAGEMENT

Our research questions and methodology presented in Chapter 3 build on a large body of related work examining semantic versioning and technical lag.

7.1.1 *Semantic Versioning*

While semantic versioning does have a precise syntactic specification [87], the semantics of what counts as backwards-compatible are not formally defined. Tooling, including NPM, generally does not enforce how developers make use of semantic versioning in practice. Choices of semantic versioning usage impact speed of distribution of packages, technical lag, stability, developer frustration, and more. Developer interviews in 2015 conducted by Bogart et al. [9] in the NPM and CRAN ecosystems found that developers try to use semantic versioning, but are not always aware of its implications and generally find dependency management exhausting. More concretely, Raemaekers et al. [90] [90] found that in 2006–2011, Maven developers often introduced binary incompatible changes within supposedly non-breaking semver updates. Wittern et al. [106] studied dependencies between packages in NPM, and found that the number of dependencies between packages is increasing over time, and observed the frequencies of version constraint types in 2016. Dietrich et al. [28] then observed how version constraint type frequencies have changed over time, at the project level. Examining version constraint evolution at the full-ecosystem level allows for an evaluation based on "wisdom of the crowds." Decan et al. [24] perform an analysis of dependency constraints at the ecosystem level for Cargo, NPM, Packagist and Rubygems. Focusing only on a single ecosystem (NPM), we validate Decan et al's findings in Chapter 3, and perform a much deeper analysis of the dataset. Our study also examines the frequencies of released update types, which enables us to draw important implications about the diffusion of security updates.

7.1.2 *Technical Lag*

Many pieces of prior work attempt to analyze the propagate of updates to downstream packages, and how out-of-date the dependencies of a project typically are. Gonzalez-Barahona et al. [39] define the measure of "technical lag", which analyzes how far out-of-date a package's dependencies are relative to more recently released versions, which has since been been further studied in the context of NPM [25, 112, 114]. In addition, the concept of technical lag is specialized to the analysis of the propagation of security patches or vulnerabilities in further work [15, 26, 113].

Calculating technical lag is difficult, and prior works have attempted to simulate the dependencies that would have been resolved at different points in time. Some of these works do not consider transitive dependencies [25, 26], which is concerning as transitive dependencies typically represent the majority of a package's dependencies in NPM. Others have followed up by considering transitive dependencies [113, 114]. Liu et al. [60] introduce *DTResolver*, a custom dependency solving algorithm that more closely matches the behavior of NPM. However, the authors' evaluation of DTResolver found that it only matched NPM's behavior when building dependency trees for 90.58% of 15,673 libraries [60]. Our exploration of NPM's dependency resolution semantics in Chapters 4 and 5 showed a variety of corner cases in which NPM's algorithm will select unexpected versions for dependencies in order to unify versions. Particularly when resolving transitive dependencies, the error introduced by an incorrect approximation of NPM's resolution semantics compounds. Compared to *all* prior work that we are aware of in studying technical lag in the NPM ecosystem, ours is the *only* study to use NPM itself to resolve historical dependencies. Our tools and dataset are available to allow others to employ this methodology [82].

7.2 SOLVER-BASED PACKAGE MANAGEMENT

The version selection problem was first shown to be NP-complete and encoded as a SAT and Constraint Programming (CP) problem by Di Cosmo et al. [27, 65] in 2005. This early work led to the Mancoosi project, which developed the idea of a modular package manager with customizable solvers [2, 3]. This work centers around the Common Upgradeability Description Format (CUDF), an input format for front-end package managers to communicate with back-end solvers.

CUDF facilitated the development of solver *implementations* using Mixed-Integer Linear Programming, Boolean Optimization, and Answer Set Programming [5, 36, 68], and many modern Linux distributions have adopted CUDF-like approaches [1]. OPIUM [101] examined the use of ILP with weights to minimize the number of bytes downloaded or the total number of packages installed.

While package managers have their roots in Linux distributions, they have evolved considerably since the early days. Modern language ecosystems have evolved their own package managers [8, 13, 93, 104], with solver requirements distinct from those of a traditional Linux distribution. Distribution package managers typically manage only a single, global installation of each package, while language package managers are geared more towards programmers and allow *multiple* installations of the same software package.

For the most part, language ecosystems have avoided using complete solvers. As we have found in our implementation, solvers are complex and interfacing with them effectively is more challenging compared to implementing a greedy algorithm. Even on the Linux distribution side, so-called *functional* Linux distributions [20, 29] eschew solving altogether, opting instead to focus on reproducible configurations maintained by humans. Most programmers do not know how to use solvers effectively, and fast, high-quality solver implementations do not exist for new and especially interpreted languages. Moreover, package managers are now fundamental to software ecosystems, and most language communities prefer to write and maintain their core tooling in their own language.

Despite this, developers are starting to realize the need for completeness and well defined dependency resolution semantics [1]. The Python community, plagued by inconsistencies in resolutions done by PIP has recently switched to a new resolver with a proper solver [89]. Dart now uses a custom CDCL SAT solver called `PubGrub` [104], and Rust's Cargo [13] package manager is moving towards this approach [88]. However, these solvers use ad hoc techniques baked into the implementations to produce desirable solutions, such as exploring package versions sorted by version number. These are not guaranteed to be optimal, and it is unclear how to add or modify objectives to these types of solvers. In contrast, PacSolve makes two new innovations: PacSolve allows for a declarative specification of multiple prioritized optimization objectives, and PacSolve changes the problem representation from prior works' boolean-variable-per-dependency representation based on SAT solving to PacSolve's symbolic graph representation (Section 4.4) based on SMT constraints. Wang et al [102] build SMARTPIP, a system similar to PacSolve for solving Python dependencies using SMT, and incorporate

an optimization objective that prioritizes unification with local, already instealled dependencies. An interesting direction would be to see if SMARTPIP is expressible in the PACSOLVE framework, so that the space savings they achieve can be applied easily to other ecosystems.

Solvers themselves are becoming more accessible through tools like Rosette [100], which makes features of the Z3 [22] SMT solver accessible within regular Racket [30] code, and which we leverage to implement PACSOLVE. Spack [35] makes complex constraints available in a Python DSL, and implements their semantics using Answer Set Programming [33, 37]. Similarly to PACSOLVE, Spack supports multi-objective optimization (backed by ASP), and uses it to find high-quality dependency solutions in HPC environments [34]. APT is moving towards using Z3 to implement more sophisticated dependency semantics [51].

The goal of PACSOLVE is to further separate concerns away from package manager developers. PACSOLVE focuses on *consistency* criteria and formalizes the *guarantees* that can be offered by package solvers. NPM [93]'s tree-based solver avoids the use of an NP-complete solver by allowing multiple, potentially *inconsistent* versions of the same package in a tree. Tools like Yarn [108], NPM's audit tool [73], Dependabot [38], Snyk [95] and others [55, 86, 96, 97] attempt to answer various needs of developers by using ad hoc techniques separate from the solving phase, such as deduplication via *hoisting* [64] (Yarn), or post hoc updating of dependencies (NPM's audit tool). However, these tools run the risk of both correctness bugs and non-optimality in their custom algorithms. PACSOLVE provides the best of all these worlds. It combines the flexibility of multi-version resolution algorithms with the guarantees of complete package solvers and being able to reason about multiple optimization objectives that each speak to a need of developers, while guaranteeing a *minimal* dependency graph.

7.3 AUTOMATED DEPENDENCY REPAIR

Generally, there have been two main branches of thought for how to perform dependency repair: search-based techniques which iteratively execute programs to search for repairing solutions, and techniques which use static analysis to infer required dependencies (and possibly versions).

7.3.1 *Static Analysis Directed Dependency Repair*

DockerizeMe [47] aims to attack the problem of dependency inference. Their goal is to, given some Python code, produce a Dockerfile that installs the dependencies necessary for it to execute correctly. They do so by analyzing which modules the program attempts to import, and pairing that with offline analysis of which Python packages provide which modules. However, they do not consider dependency *versions*, only identifying which dependencies should be installed. The same authors then have a follow-up paper [48] (discussed below) which attempts to find correct versions by iterative search.

PyEGo [109] and PyCRE [14], published concurrently in 2022, both attempt to infer not only dependencies but also versions of dependencies, and in some cases system-level libraries and Python versions. Both work by performing lightweight static analysis to determine which modules are imported and which attributes are accessed within those modules, and then using that data to build constraints to guide dependency solving.

DValidator [61] does not attack the problem of dependency repair but rather attempts to build a solver that detects various dependency smells (lints) based on the dependency graph and the call graph, such as a dependency that is declared but is never actually called. However, the authors suggest that this technique likely could be repurposed to analyze whether or not a specific version of a dependency would be incorrect for the program (similar to PyEGO and PyCRE) and by extension could be used for dependency repair.

Static analysis algorithms which synthesize constraints to guide dependency solving need not be in opposition to search-based techniques like Repyro. In fact, they are orthogonal, and could be combined quite naturally. An initial static analysis tool could synthesize constraints which are necessary (but not necessarily sufficient) for successful program execution. These constraints could then be asserted within Repyro, and would reduce the size of the dependency solution space. If the program still fails to execute after that, search-based dependency repair could attempt a repair.

An interesting variant in the space of static analysis directed repair is using offline-trained probablistic models to predict if a given solution is likely to succeed or not. BuildCheck [66] trains a graph neural network to act as a probabilistic model for the likelihood that a pair of dependencies causes a build failure. BuildCheck then adds these probability estimates as optimization criteria to the solver in the Spack package manager.

7.3.2 *Search-Based Dependency Repair*

Repyro is a search-based dependency repair tool which operates by iteratively mutating dependency constraints based on feedback from solving the current constraints and executing the program. Prior work using search-based techniques has instead primarily worked by mutating dependency solutions.

DockerizeMe V2 [48] attempts to infer correct dependency versions for unpinned dependencies inferred by DockerizeMe [47] (discussed above). V2 works by performing an iterative deepening depth-first search of the dependenct solution space, guided by fault localization and upgrade compatibility statistics mined from CI builds. Unfortunately, the CI build guidance may only be useful for the most popular packages, and this technique does not find *constraint* patches, only version pinnings which fix the error.

PyDFix [70] is a tool that comes out of the BugSwam and BugsInPy defect datasets and attempts to repair dependency drift errors which have been introduced since the time the defect sample was collected. Unlike our notion of dependency repair, PyDFix does not necessarily aim to produce a working program; it aims to fix just enough dependency errors to reproduce the build status (success or a certain error message) at the time of dataset collection. Like V2, PyDFix works by an iterative patching process, in which likely responsible dependencies are localized from the error log, and then patches are generated to pin those dependencies to an old version. PyDFix can perform backtracking by enqueueing multiple patches for a single error in the queue. The PyDFix approach only uses information from the error log to guide the selection for which dependency should be mutated, but not to determine which old version to pin to. Version choice is dictated by a heuristic of installing the latest version that was available when the original log was collected. While this heuristic makes sense in the context of CI builds, in the general setting this may not work as well, since Python developers can keep dependencies in their local environments for long periods. Additionally, PyDFix also does not produce new constraints as patches, only pinned versions.

Reliabuild [67] performs version mutation to search the configuration space of the Spack package manager. Specifically, Reliabuild uses active learning, in which it interleaves dependency solution selection, evaluation of the solution, and updating a probabilistic model. As the search progresses, the probabilistic model learns more about which dependencies likely conflict with which, and higher-quality solutions are selected. At an abstract level, this is similar to the architecture of Repyro, with the probabilistic model taking the place of the Repyro constraints. One key difference is

that REPYRO leverages an off-the-shelf LLM's prior knowledge of how dependencies and error messages are related rather than constructing that on-the-fly.

8

CONCLUSION

In this dissertation, we explore the landscape of software dependency management, focusing on package management tools that are critical across modern package ecosystems. Package managers automate much of the labor involved in specify, installing, and updating dependencies, yet programmers still face significant challenges with end-to-end dependency management, including how to optimize dependencies and how to repair broken dependency solutions. This dissertation shows that software dependency specification and management can be formalized (Chapter 4), and that such a formalization allows us to build tools offering improved optimization and repair of dependencies (Chapters 5 and 6). We now review the primary contributions of this dissertation, and reflect on consequences of design decisions and opportunities for future work.

EMPIRICAL ANALYSIS OF DEPENDENCY USAGE Chapter 3 presents an empirical study of dependency usage in the NPM ecosystem, which is the largest package repository for JavaScript. We constructed a comprehensive dataset containing every package and version available on NPM, along with metadata and security advisories. Using this dataset, we investigated research questions related to how developers specify dependencies, the extent to which they utilize semantic versioning, and how well dependency updates propagate through the ecosystem.

One of the most interesting results is a potential misalignment between package updates and package consumers: on one hand, package consumers make no distinction between bug and minor updates, as virtually all constraints are either minor-flexible or fully pinned. On the other hand, package updates are statistically different between minor and bug updates with regards to which types of files are changed and their correlation with the introduction of security vulnerabilities. An immediate solution would be for developers to consider using bug-flexible constraints if they wish to be more conservative with dependency updates, but this solution would inherently be non-composable with dependents, as they would be forced to adopt the preference of their dependency. This insight suggests that package managers should provide

mechanisms for programmers to express *soft dependency solving preferences* externally from hard constraints.

A FORMAL MODEL FOR DEPENDENCY SOLVING Chapter 4 presents PacSolve, a formal, parameterized semantics for dependency solving. Existing package managers like NPM and PIP have evolved to solve dependencies in various ways, leading to differing design philosophies and tradeoffs. To better understand this, we developed a general and executable formal model called PacSolve that allows for a flexible specification of different dependency solving policies. This formalization provides a principled framework to reason about dependency management and forms the basis for the subsequent tools and optimizations discussed in later chapters.

However, PacSolve is not able to model all known dependency solving features across the wide array of available package managers. Section 4.5 discusses some of these limitations for PacSolve, and while some limitations could be addressed naturally in future work, addressing other limitations (such as virtual packages) may necessarily involve bringing the interface of PacSolve closer to that of an SMT-solver, at which point it would lose its value as a higher-level framework for reasoning about dependency solving.

A FLEXIBLE, OPTIMIZING DEPENDENCY SOLVER Chapter 5 presents MaxNPM, a complete, drop-in replacement for NPM's dependency solver which uses PacSolve as the backend solver. NPM's default solver often makes greedy choices, resulting in suboptimal solutions that include outdated or duplicated dependencies. MaxNPM uses the Max-SMT-based optimization capabilities of PacSolve to allow developers to customize their optimization preferences, such as minimizing code size or avoiding security vulnerabilities, thus providing a solution to the problem of non-composable preferences discussed in Chapter 3. Our evaluation demonstrated that MaxNPM selects newer dependencies, reduces code size, and avoids security vulnerabilities more effectively than standard NPM.

While our experiments showed that MaxNPM generally solved dependency only a bit slower than NPM, this may not scale when running on large numbers (1000s) of dependencies. Specifically, the implementation of PacSolve requires MaxNPM to call it with the entire dependency graph eagerly evaluated, which requires MaxNPM to make potentially a large number of network requests up-front to build the full graph. An alternate approach could be to re-implement PacSolve to enable lazy evaluation of the dependency graph so that subgraphs which are known to be either unsatisfiable or

suboptimal (leveraging optimization functions which are linear over nodes) need not be fetched over the network.

AN AUTOMATED DEPENDENCY REPAIR FRAMEWORK Finally, Chapter 6 addresses the problem of dependency drift, where dependency constraints that once worked may later cause software failures due to changes in the ecosystem. We propose REPYRO, an automated Python dependency repair framework built on PACSOLVE. REPYRO leverages large language models (LLMs) to suggest constraint fixes and uses PACSOLVE's optimization capabilities to heuristically prioritize solutions which are likely better. Our ablation study on datasets of Python Gists and Jupyter notebooks showed that by combining LLM-based techniques with Max-SMT-based optimization we are able to achieve dependency repairs in fewer iterations compared to those techniques separately. Additionally, the design of REPYRO allows for the generation of modified dependency constraints instead of specific versions (offering better interpretability and integration with other tools), and for the identification of failures that are due to non-dependency issues.

A

PROMPT TEMPLATE USED IN REPYRO

````
You are an expert at fixing dependency related bugs in Python. Your task is to write a constraint
that fixes the given error while minimally constraining the solution space.

First, I will describe the syntax of constraints. The constrains are written in an embedded DSL
in Python. The DSL is a subset of Python that only includes the following constructs:
- `Python(spec: str)`: This constrains which versions of Python are allowed. The `spec` argument
is a string that can be any valid pip-style version specifier. For example, `Python(spec='>=3.6')`
would allow any version of Python 3.6 or later.
- `Installed(package: str, spec: str)`: This constrains which versions of a package are allowed.
The `package` argument is a string that is the name of the package. The `spec` argument is a string
that can be any valid pip-style version specifier. For example, `Installed('numpy', spec='<1.24')`
would allow any version of numpy strictly older than 1.24.
- `Or(A, B)`: This is a logical OR between two constraints. For example, `Or(Python(spec='<3'),
Installed('numpy', spec='>=1.18'))` would allow any version of Python 2 OR any version of numpy
1.18 or newer.
- `And(A, B)`: This is a logical AND between two constraints. For example, `And(Python(spec='>=3.6'),
Installed('numpy', spec='>=1.18'))` would allow any version of Python 3.6 or newer AND any version of
numpy 1.18 or newer.
You MUST answer in exactly this format:
```
<constraint>
```
where `<constraint>` is a SINGLE valid constraint written in the DSL. If you do not follow this
format, I will not be able to understand your answer. If you write multiple lines, I will not be
able to understand your answer. If you write anything other than a valid constraint, I will not be
able to understand your answer. If you use any operators or syntax not described above, I will not
be able to understand your answer.

Alternatively, if you believe that the error is not fixable by modifying Python dependencies, you
can write:
```
NOT_DEP_ERROR
```

I'm now going to give you a series of example error messages and correct responses.

Example Error 1:
````

```
File "/pyenv_runner/context/main.py", line 120
print 'Player data has been successfully converted.'
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print(...)?
```

Example Constraint 1:
```
Python(spec='<3')
```

Explanation of Example 1:
The error message indicates that the code is using Python 2 syntax for the `print` statement,
so it crashes with Python 3.
The constraint `Python(spec='<3')` would disallow Python 3 and only allow Python 2,
which would fix the error.

Example Error 2:
```
Traceback (most recent call last):
File "/pyenv_runner/context/main.py", line 3, in <module>
print(np.broadcast_shapes((1, 2), (3, 1)))
File "/home/pinckney/dependency-repair/tmp_venv/lib/python3.9/site-packages/numpy/__init__.py",
line 214, in __getattr__
raise AttributeError("module {{!r}} has no attribute "
AttributeError: module 'numpy' has no attribute 'broadcast_shapes'
```

Example Constraint 2:
```
Installed('numpy', spec='>=1.20')
```

Explanation of Example 2:
The error message indicates that the code is using a function `broadcast_shapes`
that was added in numpy 1.20.
Therefore, the code requires numpy 1.20 or newer, so the constraint
`Installed('numpy', spec='>=1.20')` would fix the error.

Example Error 3:
```
```

```
Traceback (most recent call last):
File "main.py", line 160, in <module>
    main()
File "main.py", line 156, in main
    get_events_list()
File "main.py", line 132, in get_events_list
    data = json.loads(get_calendar_list())
File "main.py", line 97, in get_calendar_list
    authorization_code = retrieve_authorization_code()
File "main.py", line 52, in retrieve_authorization_code
    Popen(["open", url])
File "/pyenv/versions/2.7.18/lib/python2.7/subprocess.py", line 394, in __init__
    errread, errwrite)
File "/pyenv/versions/2.7.18/lib/python2.7/subprocess.py", line 1047, in _execute_child
    raise child_exception
OSError: [Errno 2] No such file or directory
```

Example Response 3:
```
NOT_DEP_ERROR
```

Explanation of Example 3:
The error message indicates that the "open" command is not available in the environment. This is not an error that can be resolved through modifying Pip dependencies, so the correct response is NOT_DEP_ERROR.

The error message which you need to fix is:
```
{{ stderr }}
```

BIBLIOGRAPHY

[1] Pietro Abate, Roberto Di Cosmo, Georgios Gousios, and Stefano Zacchiroli. "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.

[2] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. "Dependency solving: a separate concern in component evolution management." In: *Journal of Systems and Software* 85.10 (2012), pp. 2228–2240.

[3] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. "A modular package manager architecture." In: *Information and Software Technology* 55.2 (2013). Special Section: Component-Based Software Engineering (CBSE), 2011, pp. 459–474. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2012.09.002. URL: http://www.sciencedirect.com/science/article/pii/S0950584912001851.

[4] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. "Why Do Developers Use Trivial Packages? An Empirical Case Study on Npm." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 385–395. ISBN: 9781450351058. DOI: 10.1145/3106237.3106267. URL: https://doi.org/10.1145/3106237.3106267.

[5] Josep Argelich, Daniel Le Berre, Inês Lynce, João P. Marques Silva, and Pascal Rapicault. "Solving Linux Upgradeability Problems Using Boolean Optimization." In: *Proceedings First International Workshop on Logics for Component Configuration, LoCoCo 2010, Edinburgh, UK, 10th July 2010*. Ed. by Inês Lynce and Ralf Treinen. Vol. 29. EPTCS. 2010, pp. 11–22. DOI: 10.4204/EPTCS.29.2. URL: https://doi.org/10.4204/EPTCS.29.2.

[6] Szele Balint. *The Small Files Problem*. https://blog.cloudera.com/the-small-files-problem/. Accessed Mar 13 2023. 2009.

[7] *Beautiful Soup Differences - Soup Sieve*. https://facelessuser.github.io/soupsieve/differences/. 2024.

[8] Ian Bicking. *pip: Package Install tool for Python*. github.com/pypa/pip. Apr. 2011.

[9] Christopher Bogart, Christian Kästner, and James Herbsleb. "When It Breaks, It Breaks: How Ecosystem Developers Reason about the Stability of Dependencies." In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. 2015, pp. 86–89. DOI: 10.1109/ASEW.2015.21.

[10] Manfred Broy and Martin Wirsing. "On the algebraic specification of nondeterministic programming languages." In: *Colloquium on Trees in Algebra and Programming*. Springer. 1981, pp. 162–179.

[11] Ben Caller. *Security Advisory: Regular Expression Denial of service (ReDoS) in npm/ssri*. 2021. URL: https://doyensec.com/resources/Doyensec_Advisory_ssri_redos.pdf.

[12] *The Cargo Book: Renaming Dependencies in `cargo.toml`*. https://doc.rust-lang.org/cargo/reference/specifying-dependencies.html#renaming-dependencies-in-cargotoml. 2022.

[13] *Cargo: The Rust package manager*. Online. https://github.com/rust-lang/cargo. Mar. 2014.

[14] Wei Cheng, Xiangrong Zhu, and Wei Hu. "Conflict-aware inference of python compatible runtime environments with domain knowledge graph." In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 451–461. ISBN: 9781450392211. DOI: 10.1145/3510003.3510078. URL: https://doi.org/10.1145/3510003.3510078.

[15] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. "Lags in the release, adoption, and propagation of npm vulnerability fixes." In: *Empirical Software Engineering* 26.3 (Mar. 2021), p. 47. ISSN: 1573-7616. DOI: 10.1007/s10664-021-09951-x. URL: https://doi.org/10.1007/s10664-021-09951-x.

[16] Md Atique Reza Chowdhury, Rabe Abdalkareem, Emad Shihab, and Bram Adams. "On the Untriviality of Trivial Packages: An Empirical Study of npm JavaScript Packages." In: *IEEE Transactions on Software Engineering* 48.8 (2022), pp. 2695–2708. DOI: 10.1109/TSE.2021.3068901.

[17] Nick Coghlan. *PEP 440 – Version Identification and Dependency Specification*. Online. https://www.python.org/dev/peps/pep-0440. Mar. 2013.

[18] *config: install-strategy*. https://docs.npmjs.com/cli/v10/using-npm/config#install-strategy. Accessed Sep 2 2024. 2024.

[19] Conda Contributors. *Conda performance*. docs.conda.io/projects/conda/en/latest/user-guide/concepts/conda-performance.html. Accessed Sep 1 2022. 2022.

[20] Ludovic Courtès and Ricardo Wurmus. "Reproducible and User-Controlled Software Environments in HPC with Guix." In: *2nd International Workshop on Reproducibility in Parallel Computing (RepPar)*. Vienne, Austria, Aug. 2015. URL: https://hal.inria.fr/hal-01161771.

[21] *CVE-2020-28502 Detail*. 2021. URL: https://nvd.nist.gov/vuln/detail/CVE-2020-28502.

[22] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver." In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[23] Erik DeBill. *Modulecounts*. http://www.modulecounts.com. Accessed Aug 5 2024. 2024.

[24] Alexandre Decan and Tom Mens. "What Do Package Dependencies Tell Us About Semantic Versioning?" In: *IEEE Transactions on Software Engineering* 47.6 (2021), pp. 1226–1240. DOI: 10.1109/TSE.2019.2918315.

[25] Alexandre Decan, Tom Mens, and Eleni Constantinou. "On the Evolution of Technical Lag in the npm Package Dependency Network." In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, pp. 404–414. DOI: 10.1109/ICSME.2018.00050.

[26] Alexandre Decan, Tom Mens, and Eleni Constantinou. "On the Impact of Security Vulnerabilities in the Npm Package Dependency Network." In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 181–191. ISBN: 9781450357166. DOI: 10.1145/3196398.3196401. URL: https://doi.org/10.1145/3196398.3196401.

[27] Roberto Di Cosmo. *EDOS deliverable WP2-D2.1: Report on Formal Management of Software Dependencies*. Tech. rep. hal-00697463. INRIA, May 2005.

[28] Jens Dietrich, David J. Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. "Dependency Versioning in the Wild." In: *Proceedings of the 16th International Conference on Mining Software Repositories*. MSR '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 349–359. DOI: 10.1109/MSR.2019.00061. URL: https://doi.org/10.1109/MSR.2019.00061.

[29] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. "Nix: A Safe and Policy-Free System for Software Deployment." In: *Proceedings of the 18th Large Installation System Administration Conference (LISA XVIII)*. LISA '04. Atlanta, GA: USENIX Association, 2004, pp. 79–92.

[30] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. "The racket manifesto." In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2015.

[31] Christopher Flynn. *PyPI Download Stats*. https://pypistats.org. Accessed Aug 5 2024. 2024.

[32] The Apache Software Foundation. *Apache Hadoop*. https://hadoop.apache.org. Accessed Mar 13 2023. 2023.

[33] Todd Gamblin. "Spack's new Concretizer: Dependency solving is more than just SAT!" In: *Free and Open source Software Developers' European Meeting (FOSDEM'20)*. Brussels, Belgium, Feb. 2020.

[34] Todd Gamblin, Massimiliano Culpo, Gregory Becker, and Sergei Shudler. "Using Answer Set Programming for HPC Dependency Solving." In: *Supercomputing 2022 (SC'22)*. Dallas, Texas, Nov. 2022.

[35] Todd Gamblin, Matthew P. LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and W. Scott Futral. "The Spack Package Manager: Bringing order to HPC software chaos." In: *Supercomputing 2015 (SC'15)*. LLNL-CONF-669890. Austin, Texas, Nov. 2015.

[36] Martin Gebser, Roland Kaminski, and Torsten Schaub. "aspcud: A Linux Package Configuration Tool Based on Answer Set Programming." In: *Electronic Proceedings in Theoretical Computer Science* 65 (Aug. 2011), pp. 12–25. ISSN: 2075-2180.

[37] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. "Potassco: The Potsdam answer set solving collection." In: *AI Communications* 24.2 (2011), pp. 107–124.

[38] GitHub. *Dependabot*. https://github.com/features/security/. 2023.

[39] Jesus M. Gonzalez-Barahona, Paul Sherwood, Gregorio Robles, and Daniel Izquierdo. "Technical Lag in Software Compilations: Measuring How Outdated a Software Deployment Is." In: *Open Source Systems: Towards Robust Practices*. Ed. by Federico Balaguer, Roberto Di Cosmo, Alejandra Garrido, Fabio Kon, Gregorio Robles, and Stefano Zacchiroli. Cham: Springer International Publishing, 2017, pp. 182–192. ISBN: 978-3-319-57735-7.

[40] George D Greenwade. "The comprehensive TEX archive network (ctan)." In: *TUGBoat* 14.3 (1993), pp. 342–351.

[41] The PostgreSQL Global Development Group. *PostgreSQL: The World's Most Advanced Open Source Relational Database*. https://www.postgresql.org. Accessed Mar 12 2023. 2023.

[42] Jason Gunthorpe. *APT User's Guide*. Online. https://www.debian.org/doc/manuals/apt-guide/. 1998.

[43] Robert Harper and Mark Lillibridge. "A type-theoretic approach to higher-order modules with sharing." In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '94. Portland, Oregon, USA: Association for Computing Machinery, 1994, pp. 123–137. ISBN: 0897916360. DOI: 10.1145/174675.176927. URL: https://doi.org/10.1145/174675.176927.

[44] Robert Harper, John C. Mitchell, and Eugenio Moggi. "Higher-order modules and the phase distinction." In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '90. San Francisco, California, USA: Association for Computing Machinery, 1989, pp. 341–354. ISBN: 0897913434. DOI: 10.1145/96709.96744. URL: https://doi.org/10.1145/96709.96744.

[45] Robert. Harper. *Practical foundations for programming languages*. eng. Cambridge ; Cambridge University Press, 2013. Chap. 44-46.

[46] E. Horton and C. Parnin. "Gistable: Evaluating the Executability of Python Code Snippets on GitHub." In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Los Alamitos, CA, USA: IEEE Computer Society, 2018, pp. 217–227. DOI: 10.1109/ICSME.2018.00031. URL: https://doi.ieeecomputersociety.org/10.1109/ICSME.2018.00031.

[47] 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.

[48] Eric Horton and Chris Parnin. "V2: Fast Detection of Configuration Drift in Python." In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 477–488. DOI: 10.1109/ASE.2019.00052.

[49] *ImportError: cannot import name TwilioRestClient*. https://stackoverflow.com/a/41036484. 2024.

[50] Dhanushka Jayasuriya. "Towards Automated Updates of Software Dependencies." In: *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. SPLASH Companion 2022. Auckland, New Zealand: Association for Computing Machinery, 2022, pp. 29–33. ISBN: 9781450399012. DOI: 10.1145/3563768.3565548. URL: https://doi.org/10.1145/3563768.3565548.

[51] Julian Andres Klode. *APT Z3 Solver Basics*. Online. https://blog.jak-linux.org/2021/11/21/apt-z3-solver-basics/. Nov. 2021.

[52] Raula Gaikovina Kula, Ali Ouni, Daniel M. German, and Katsuro Inoue. *On the Impact of Micro-Packages: An Empirical Study of the npm JavaScript Ecosystem*. 2017. DOI: 10.48550/ARXIV.1709.04638. URL: https://arxiv.org/abs/1709.04638.

[53] Michael Kutz. *Legit but Useless:Maven Version Ranges Explained*. https://michakutz.medium.com/legit-but-useless-maven-version-ranges-explained-d4ba66ac654. 2019.

[54] Patrick Lam, Jens Dietrich, and David J. Pearce. "Putting the Semantics into Semantic Versioning." In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 157–179. ISBN: 9781450381789. DOI: 10.1145/3426428.3426922. URL: https://doi.org/10.1145/3426428.3426922.

[55] Jasmine Latendresse, Suhaib Mujahid, Diego Elias Costa, and Emad Shihab. "Not All Dependencies Are Equal: An Empirical Study on Production Dependencies in NPM." In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE '22. Rochester, MI, USA: Association for Computing

Machinery, 2023. ISBN: 9781450394758. DOI: 10.1145/3551349.3556896. URL: https://doi.org/10.1145/3551349.3556896.

[56] Xavier Leroy. "Manifest types, modules, and separate compilation." In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '94. Portland, Oregon, USA: Association for Computing Machinery, 1994, pp. 109–122. ISBN: 0897916360. DOI: 10.1145/174675.176926. URL: https://doi.org/10.1145/174675.176926.

[57] Xavier Leroy. "A modular module system." In: *J. Funct. Program.* 10.3 (2000), pp. 269–303. ISSN: 0956-7968. DOI: 10.1017/S0956796800003683. URL: https://doi.org/10.1017/S0956796800003683.

[58] Patrick Lewis et al. "Retrieval-augmented generation for knowledge-intensive NLP tasks." In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS '20. Vancouver, BC, Canada: Curran Associates Inc., 2020. ISBN: 9781713829546.

[59] Barbara Liskov. "A history of CLU." In: *History of Programming Languages—II*. New York, NY, USA: Association for Computing Machinery, 1996, pp. 471–510. ISBN: 0201895021. URL: https://doi.org/10.1145/234286.1057826.

[60] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. "Demystifying the Vulnerability Propagation and Its Evolution via Dependency Trees in the NPM Ecosystem." In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 672–684. ISBN: 9781450392211. DOI: 10.1145/3510003.3510142. URL: https://doi.org/10.1145/3510003.3510142.

[61] Christian Macho, Fabian Oraze, and Martin Pinzger. "DValidator: An approach for validating dependencies in build configurations." In: *Journal of Systems and Software* 209 (2024), p. 111916. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2023.111916. URL: https://www.sciencedirect.com/science/article/pii/S0164121223003114.

[62] David MacQueen. "Modules for standard ML." In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP '84. Austin, Texas, USA: Association for Computing Machinery, 1984, pp. 198–207. ISBN: 0897911423. DOI: 10.1145/800055.802036. URL: https://doi.org/10.1145/800055.802036.

[63] David B. MacQueen. "Using dependent types to express modular structure." In: *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '86. St. Petersburg Beach, Florida: Association for Computing Machinery, 1986, pp. 277–286. ISBN: 9781450373470. DOI: 10.1145/512644.512670. URL: https://doi.org/10.1145/512644.512670.

[64] Thilo Maier. *A guide to understanding how Yarn hoists dependencies and handles conflicting packages*. Online. https://maier.tech/posts/a-guide-to-understanding-how-yarn-hoists-dependencies-and-handles-conflicting-packages. Nov. 2021.

[65] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. "Managing the Complexity of Large Free and Open Source Package-Based Software Distributions." In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. 2006, pp. 199–208.

[66] Harshitha Menon, Daniel Nichols, Abhinav Bhatele, and Todd Gamblin. "Learning to Predict and Improve Build Successes in Package Ecosystems." In: *2024 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2024.

[67] Harshitha Menon, Konstantinos Parasyris, Tom Scogland, and Todd Gamblin. "Searching for High-Fidelity Builds Using Active Learning." In: *Proceedings of the 19th International Conference on Mining Software Repositories*. MSR '22. Pittsburgh, PA, USA: IEEE Press, 2022.

[68] Claude Michel and Michel Rueher. "Handling software upgradeability problems with MILP solvers." In: *Proceedings First International Workshop on Logics for Component Configuration, LoCoCo 2010, Edinburgh, UK, 10th July 2010*. Ed. by Inês Lynce and Ralf Treinen. Vol. 29. EPTCS. 2010, pp. 1–10. DOI: 10.4204/EPTCS.29.1. URL: https://doi.org/10.4204/EPTCS.29.1.

[69] John C. Mitchell and Gordon D. Plotkin. "Abstract types have existential type." In: *ACM Trans. Program. Lang. Syst.* 10.3 (July 1988), pp. 470–502. ISSN: 0164-0925. DOI: 10.1145/44501.45065. URL: https://doi.org/10.1145/44501.45065.

[70] 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*. ISSTA 2021. Virtual, Denmark: Association for Computing Machinery, 2021, pp. 439–451. ISBN: 9781450384599. DOI: 10.1145/3460319.3464797. URL: https://doi.org/10.1145/3460319.3464797.

[71] Sven Neumann. *Similar Files are Collapsed (breaking react-bootstrap)*. github.com/parcel-bundler/parcel/issues/3523. Accessed Aug 29 2022. 2019.

[72] NPM. *semver(1) – The semantic versioner for npm*. https://github.com/npm/node-semver. 2022.

[73] NPM. *npm-audit*. https://docs.npmjs.com/cli/v9/commands/npm-audit. 2023.

[74] NPM and Contributors. *package.json. overrides*. https://docs.npmjs.com/cli/v9/configuring-npm/package-json#overrides. Accessed Mar 10 2023. 2023.

[75] NPM and Contributors. *registry-follower-tutorial*. https://github.com/npm/registry-follower-tutorial. Accessed Mar 12 2023. 2023.

[76] Kristen Nygaard and Ole-Johan Dahl. "The development of the SIMULA languages." In: *SIGPLAN Not.* 13.8 (Aug. 1978), pp. 245–272. ISSN: 0362-1340. DOI: 10.1145/960118.808391. URL: https://doi.org/10.1145/960118.808391.

[77] Lina Ochoa, Thomas Degueule, and Jean-Rémy Falleri. "BreakBot: Analyzing the Impact of Breaking Changes to Assist Library Evolution." In: *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. ICSE-NIER '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 26–30. ISBN: 9781450392242. DOI: 10.1145/3510455.3512783. URL: https://doi.org/10.1145/3510455.3512783.

[78] *package.json*. https://docs.npmjs.com/cli/v9/configuring-npm/package-json#dependencies. Accessed Jan 20 2023. 2023.

[79] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. "Understanding and improving the quality and reproducibility of Jupyter notebooks." In: *Empirical Software Engineering* 26.65 (May 2021). ISSN: 1573-7616. DOI: 10.1007/s10664-021-09961-9. URL: https://doi.org/10.1007/s10664-021-09961-9.

[80] D. Pinckney, F. Cassano, A. Guha, and J. Bell. "A Large Scale Analysis of Semantic Versioning in NPM." In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 485–497. DOI: 10.1109/MSR59073.2023.00073. URL: https://doi.ieeecomputersociety.org/10.1109/MSR59073.2023.00073.

[81] Donald Pinckney and Federico Cassano. *PacSolve*. https://github.com/donald-pinckney/pacsolve. 2023.

[82] Donald Pinckney, Federico Cassano, Arjun Guha, and Jonathan Bell. *Artifact For A Large Scale Analysis of Semantic Versioning in NPM*. Zenodo, Jan. 2023. DOI: 10.5281/zenodo.7552551. URL: https://doi.org/10.5281/zenodo.7552551.

[83] Donald Pinckney, Federico Cassano, Arjun Guha, Jonathan Bell, Massimiliano Culpo, and Todd Gamblin. *Artifact for Flexible and Optimal Dependency Management via Max-SMT*. https://doi.org/10.5281/zenodo.7554407. 2023.

[84] Donald Pinckney, Federico Cassano, Arjun Guha, Jonathan Bell, Massimiliano Culpo, and Todd Gamblin. "Flexible and Optimal Dependency Management via Max-SMT." In: *Proceedings of the 2023 International Conference on Software Engineering*. ICSE. 2023.

[85] Contributors of pnpm. *pnpm vs npm*. https://pnpm.io/pnpm-vs-npm. Accessed Sep 1 2024. 2024.

[86] Serena Elisa Ponta, Wolfram Fischer, Henrik Plate, and Antonino Sabetta. "The Used, the Bloated, and the Vulnerable: Reducing the Attack Surface of an Industrial Application." In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2021, pp. 555–558. DOI: 10.1109/ICSME52107.2021.00056.

[87] Tom Preston-Werner and Contributors. *Semantic Versioning 2.0.0*. https://semver.org. Accessed Mar 9 2023. 2023.

[88] *PubGrub version solving algorithm implemented in Rust*. Online. https://github.com/pubgrub-rs/pubgrub. 2020.

[89] Python Software Foundation. *New pip resolver to roll out this year*. Online. https://pyfound.blog pip-resolver-to-roll-out-this-year.html. Mar. 2020.

[90] Steven Raemaekers, Arie van Deursen, and Joost Visser. "Semantic Versioning versus Breaking Changes: A Study of the Maven Repository." In: *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. 2014, pp. 215–224. DOI: 10.1109/SCAM.2014.30.

[91] Ryan Roemer. *Finding and fixing duplicates in webpack with Inspectpack*. formidable.com/blog/2018/finding-webpack-duplicates-with-inspectpack-plugin/. Accessed Aug 29 2022. 2022.

[92] SchedMD and Contributors. *Slurm Workload Manager – Documentation*. https://slurm.schedmd.com. Accessed Mar 12 2023. 2023.

[93] Isaac Z. Schlueter. *NPM*. Online. https://github.com/npm/npm. Sept. 2009.

[94] Adriana Sejfia and Max Schäfer. "Practical Automated Detection of Malicious Npm Packages." In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1681–1692. ISBN: 9781450392211. DOI: 10.1145/3510003.3510104. URL: https://doi.org/10.1145/3510003.3510104.

[95] *Snyk Open Source*. https://snyk.io/product/open-source-security-management/. Accessed Jan 20 2023. 2023.

[96] César Soto-Valero, Thomas Durieux, and Benoit Baudry. "A Longitudinal Analysis of Bloated Java Dependencies." In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 1021–1031. ISBN: 9781450385626. DOI: 10.1145/3468264.3468589. URL: https://doi.org/10.1145/3468264.3468589.

[97] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. "A comprehensive study of bloated dependencies in the Maven ecosystem." In: *Empirical Software Engineering* 26 (2021). DOI: https://doi.org/10.1007/s10664-020-09914-8. URL: https://link.springer.com/article/10.1007/s10664-020-09914-8#citeas.

[98] *The Comprehensive Perl Archive Network*. https://www.cpan.org. 2024.

[99] *The Comprehensive R Archive Network*. https://cran.r-project.org. 2024.

[100] Emina Torlak and Rastislav Bodik. "Growing Solver-Aided Languages with Rosette." In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 135–152. ISBN: 9781450324724. DOI: 10.1145/2509578.2509586. URL: https://doi.org/10.1145/2509578.2509586.

[101] Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. "OPIUM: Optimal Package Install/Uninstall Manager." In: *Proceedings of the 29th International Conference on Software Engineering*. ICSE '07. USA: IEEE Computer Society, 2007, pp. 178–188. ISBN: 0769528287.

[102] Chao Wang, Rongxin Wu, Haohao Song, Jiwu Shu, and Guoqing Li. "smartPip: A Smart Approach to Resolving Python Dependency Conflict Issues." In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE '22. Rochester, MI, USA: Association for Computing

Machinery, 2023. ISBN: 9781450394758. DOI: 10.1145/3551349.3560437. URL: https://doi.org/10.1145/3551349.3560437.

[103] Webpack Contributors. *Mark the file as side-effect-free*. webpack.js.org/guides/tree-shaking/#mark-the-file-as-side-effect-free. Accessed Aug 29 2022. 2022.

[104] Natalie Weizenbaum. *PubGrub: Next-Generation Version Solving*. https://medium.com/@nex3/2fb6470504f. Apr. 2018.

[105] N. Wirth. "Modula: A language for modular multiprogramming." In: *Software: Practice and Experience* 7.1 (1977), pp. 1–35. DOI: https://doi.org/10.1002/spe.4380070102. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380070102. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380070102.

[106] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. "A Look at the Dynamics of the JavaScript Package Ecosystem." In: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR '16. Austin, Texas: Association for Computing Machinery, 2016, pp. 351–361. ISBN: 9781450341868. DOI: 10.1145/2901739.2901743. URL: https://doi.org/10.1145/2901739.2901743.

[107] J. Yang and D. Evans. "Automatically inferring temporal properties for program evolution." In: *15th International Symposium on Software Reliability Engineering*. 2004, pp. 340–351. DOI: 10.1109/ISSRE.2004.11.

[108] *Yarn: Yet Another Resource Negotiator (Javascript package manager)*. https://github.com/yarnpkg/ 2022.

[109] Hongjie Ye, Wei Chen, Wensheng Dou, Guoquan Wu, and Jun Wei. "Knowledge-based environment dependency inference for python programs." In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1245–1256. ISBN: 9781450392211. DOI: 10.1145/3510003.3510127. URL: https://doi.org/10.1145/3510003.3510127.

[110] Andy B. Yoo, Morris A. Jette, and Mark Grondona. "SLURM: Simple Linux Utility for Resource Management." In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. ISBN: 978-3-540-39727-4.

[111] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. "What Are Weak Links in the Npm Supply Chain?" In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 331–340. ISBN: 9781450392266. DOI: 10.1145/3510457.3513044. URL: https://doi.org/10.1145/3510457.3513044.

[112] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. "An Empirical Analysis of Technical Lag in npm Package Dependencies." In: *New Opportunities for Software Reuse*. Ed. by Rafael Capilla, Barbara Gallina, and Carlos Cetina. Cham: Springer International Publishing, 2018, pp. 95–110. ISBN: 978-3-319-90421-4.

[113] Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. "On the impact of security vulnerabilities in the npm and RubyGems dependency networks." In: *Empirical Software Engineering* 27.5 (May 2022), p. 107. ISSN: 1573-7616. DOI: 10.1007/s10664-022-10154-1. URL: https://doi.org/10.1007/s10664-022-10154-1.

[114] Ahmed Zerouali, Tom Mens, Jesus Gonzalez-Barahona, Alexandre Decan, Eleni Constantinou, and Gregorio Robles. "A formal framework for measuring technical lag in component repositories — and its application to npm." In: *Journal of Software: Evolution and Process* 31.8 (2019). e2157 smr.2157, e2157. DOI: https://doi.org/10.1002/smr.2157. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2157. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2157.