AGH University of Krakow

**THE FACULTY OF COMPUTER SCIENCE,
ELECTRONICS AND TELECOMMUNICATIONS**

Master's thesis

Software Provenance Assurance through Reproducible Builds

*Potwierdzanie autentyczności oprogramowania
poprzez powtarzalność kompilacji*

Keywords: software supply chain threats, reproducible builds, software provenance, software packaging, npm Registry

Author:	Wojciech Kosior
Major:	Cybersecurity
Supervisor:	dr hab. inż. Piotr Pacyna, prof. AGH

Kraków, 2025

Acknowledgements

I could have decided not to enroll for the MSc course in the first place. But, having experienced some failures in my career, I started asking what God would like me to do. I recalled that even though at various moments it was unpleasant to be a student, university was a place that suited me more that places I have been to afterwards. I assumed that maybe – just maybe – God did not want me to succeed anywhere else because He wants me to be happy here, in the academia. And so I am, finishing this unusual piece of research. Thank God.

I also want to thank my wife, Joanna, who supported me even though she regularly had to suffer listening about my boring computer topics. I thank my father, who at times seems to care for my successful defence even more than I do.

Finally, I could not forget about my supervisor who – luckily for me – was corageous enough to undertake supervising a thesis in the software supply chain field, which few students pursued before. Thank you!

Contents

List of Figures

List of Listings

List of Tables

Abstract

Software faces risk of contamination on multiple stages of its creation and distribution. One such stage is software's build, where its initial form – the source code – is transformed into a form suitable for distribution. A build is called reproducible when it yields bit-to-bit identical outputs when repeated. The reproducible build can be secured by repeating it on multiple infrastructures and comparing the outputs. This decreaces the risk of the software getting contaminated through the build infrastructure used. Certain software projects – in particular, the Debian operating system – already levarage reproducible builds as a security tool. Meanwhile, several software ecosystems rely on repositories whose packages cannot be reliably rebuilt and tested for reproducibility. An example is a popular repository called npm Registry. The software available through the npm Registry gets included in reproducibility-focused distributions like Debian at slow pace. A great number and complexity of dependency relations between npm packages were hypothesized to be the primary hindrances to this process. In this work, a statistical analysis of the npm ecosystem was performed and existing approaches to reproducibility in the context of dependency resolution were identified. Additionally, alternative approaches were proposed that would allow for easier packaging of npm software while making reproducibility achievable. To verify several stated hypotheses, an experiment was performed, where builds of the most popular npm projects were attempted. Projects were built multiple times to test what subset of their npm Registry dependencies can be removed without causing the build to fail. The complexity and size of project's minimal dependency tree was found not to be related to the likelihood of the project having a corresponding Debian package. The results lead to conclusion that even numerous and complex dependency relations can be handled in existing reproducibility-focused software distributions. This means that employing the proposed new approaches is not necessary to apply reproducibility to npm software in the future. It also means that the inclusion of npm software in reproducibility-focused software distributions is mostly hindered by other factors that need to be countered. These were also pointed at the end of this work.

Streszczenie

Bezpieczeństwo oprogramowania może zostać zagrożone na różnych etapach jego tworzenia i dystrybucji. Jednym z nich jest szeroko rozumiana kompilacja oprogramowania, gdzie jego pierwotna postać – kod źródłowy – jest przekształcana do postaci odpowiedniej dla dystrybucji. Proces kompilacji nazywamy powtarzalnym, jeśli przy wielokrotnym przeprowadzeniu daje wyniki bit do bitu identyczne. Powtarzalny proces kompilacji może zostać zabezpieczony poprzez przeprowadzenie go na różnych infrastrukturach i porównanie wyników. Zmniejsza to ryzyko zanieczyszczenia kodu oprogramowania przez użytą infrastrukturę. Pewne oprogramowanie – w szczególności system Debian – już wykorzystuje powtarzalność jako narzędzie bezpieczeństwa. Jednocześnie, niektóre ekosystemy oprogramowania polegają na repozytoriach, których pakiety nie mogą być w sposób niezawodny przekompilowane i sprawdzone pod kątem powtarzalności. Przykładem jest popularne repozytorium o nazwie npm Registry. Oprogramowanie dostępne przez npm Registry jest też, aczkolwiek w wolnym tempie, włączane do dystrybucji typu Debian dbających o powtarzalność kompilacji. Według postawionej hipotezy to duża liczba i złożoność relacji zależności między pakietami npm są głównymi utrudnieniami w tym procesie. W ramach pracy została wykonana analiza statystyczna ekosystemu npm oraz zostały zidentyfikowane istniejące podejścia do powtarzalności w kontekście procesu rozwiązywania zależności. Dodatkowo, zostały zaproponowane alternatywne podejścia, w których pakowanie oprogramowania npm miałoby być łatwiejsze, a powtarzalność byłaby wciąż osiągalna. Dla zweryfikowania postawionych hipotez przeprowadzony został eksperyment – próba kompilacji najpopularniejszych projektów npm. Projekty kompilowano wielokrotnie, sprawdzając, jaka część ich zależności z npm Registry może zostać usunięta z zachowaniem powodzenia procesu kompilacji. Złożoność i rozmiar minimalnego drzewa zależności projektu okazały się nie być powiązane z prawdopodobieństwiem istnienia odpowiadającego pakietu w Debianie. Wyniki prowadzą do wniosku, że istniejące dystrubucje oprogramowania dbające o powtarzalność mogą sobie poradzić także z licznymi i złożonymi relacjami zależności. Oznacza to, że wprowadzenie zaproponowanych nowych podejść nie jest konieczne, żeby można było w przyszłości zastosować powtarzalność do oprogramowania npm. Oznacza to też, że włączanie oprogramowania npm do dystrybucji oprogramowania dbających o powtarzalność jest w głównej mierze powstrzymywane przez inne czynniki wymagające zwalczenia. Zostały one wskazane pod koniec pracy.

1. Introduction

Most products of modern industry are composed or made using a number of half-products and tools. These tend to come from different producers, who themselves rely on their suppliers. Such half-products might be produced with violations of human rights, unecologically, without meeting certain quality standards, or with patent violations. The assembly from half-products also creates an opportunity for deliberate sabotage on part of the supplier. So far businesses have not always successfully mitigated these threats, which later reverberated in many ways.

Just as a design is often used to manufacture physical products, code written in a programming language is used to produce software in its target form, e.g., an executable, a firmware image, or a collection of files. This process of producing software in its target form can be referred to as software **build**. Items resulting from it are **build outputs**. A software build process overview is presented in Figure 1.1. Depending on the programming languages and technologies in use, the build might encompass different actions, e.g., macro processing, compilation, linking, bundling, or compressed archive creation. It can also utilize a multitude of different tools. Adjusting the build process to meet certain requirements is often a complex task that requires understanding the workings of various tools involved. In some software distribution projects, hundreds of lines of scripts are maintained to allow proper building of a single piece of software written by another party. Builds can even involve applying changes to upstream code, often through the use of patches, i.e., machine-readable files describing changes to project code.

Similarly to physical products, software is made using preexisting elements delivered by other parties. These are often called **dependencies**. They include

- runtime dependencies – components to be distributed inside or alognside the final program and used during its execution, for example reusable modules of code called libraries or special fonts, and

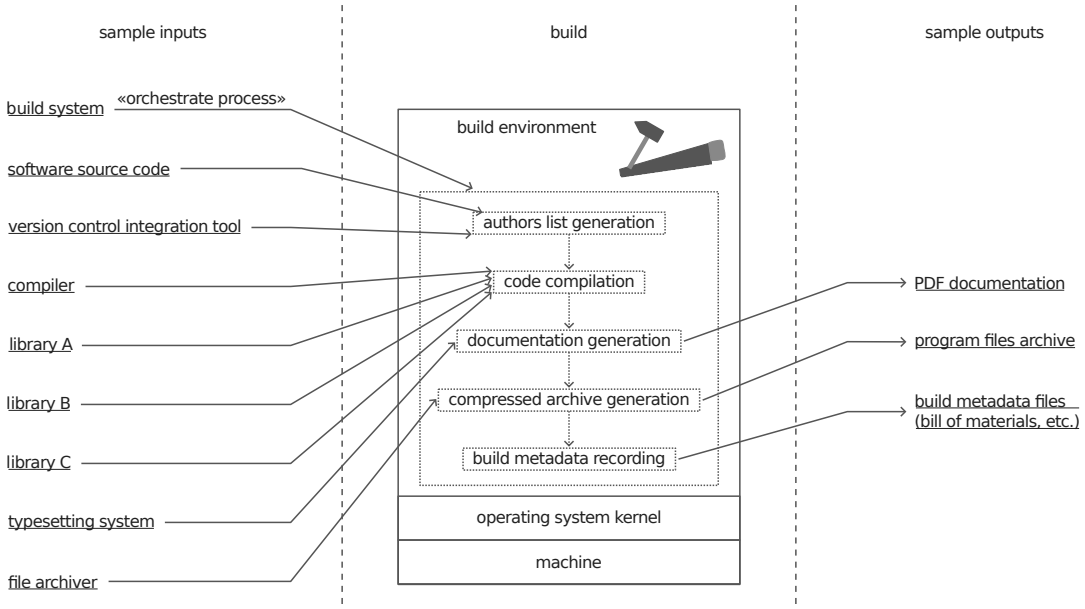

Figure 1.1: Overview of a sample software build process.

- development dependencies – elements not needed during program's execution but useful to its developers, furter classifiable as
  - build dependencies – compilers, linkers, test frameworks, etc. needed in the build process, typically able to function non-intereactively and invoked through some layer of automation, sometimes categorized further, for example into native and non-native or into build and test dependencies, and
  - other development tools – tools to work with software that are not needed in the actual build process, more often manually-operated, like IDEs[1] and their plugins, debuggers, or linters.

1.1 Problem formulation

If either an external dependency suffers from contamination, the infrastructure handling the build is compromised, or the organization or individuals attempt a sabotage, then a backdoor or other security vulnerability can be implanted in the software being built. In a basic setting, each dependency, the infrastructure, and the organization are all single points of failure. The last two of these points can be secured through additional build outputs ver-

[1] Integrated Development Environments

ification that utilizes software reproducibility. This work aims at exploiting reproducibility to secure software's build process, with special focus on eliminating the gaps that would leave single points of failure.

A reproducible build is one that produces the same, bit-to-bit identical outputs when repeated. For example, the resulting program executables are bit-to-bit identical. This concept assumes that a set of **build inputs** with the same contents is used in every repetition. E.g., program sources and dependencies in exact same versions are used. As a consequence, one prerequisite of reproducibility is **hermeticity** – the quality of a build process that depends exclusively on a set of predefined dependencies. A hermetic build must not depend on changeable network resources nor on machine's installed software other than build's formal inputs. Hermeticity is usually ensured by performing software build inside a minimal, isolated environment, often a container utilizing Linux namespaces.

Multiparty verification of build's reproducible output can help increase confidence that built software is not contaminated due to compromise of infrastructure underpinning the build environment nor malicious actions of infrastructure operators. The verification shall be unsuccessful if the contamination is present in an output of one build and not in those of the others. The overviews of successful and unsuccessful verification performed by end user – a scheme that does not create unnecessary single points of failure – are presented in Figures 1.2 and 1.3, respectively. Contamination is represented by a frowning face. The extra confidence coming from verification can empower both software vendors willing to reduce the risk of distributing compromised code and software consumers wanting to secure their operations.

Single-party verification is also applicable if only the infrastructure threats are considered. Meanwhile, the party itself retains the ability to undetectably compromise software builds, i.e., implant backdoors. Figure 1.4 depicts an occurrence of such compromise while single-party verification of build's reproducible output is taking place. Contamination is represented by a frowning face.

In addition to the above, just as reproducible builds performed by a single organization are insufficient to protect from contamination introduced deterministically by the organization, reproducible builds performed on a single infrastructure would be insufficient to protect from contamination spreading deterministically from that infrastructure.

For software to be secured with reproducible builds, its build process has to gain the

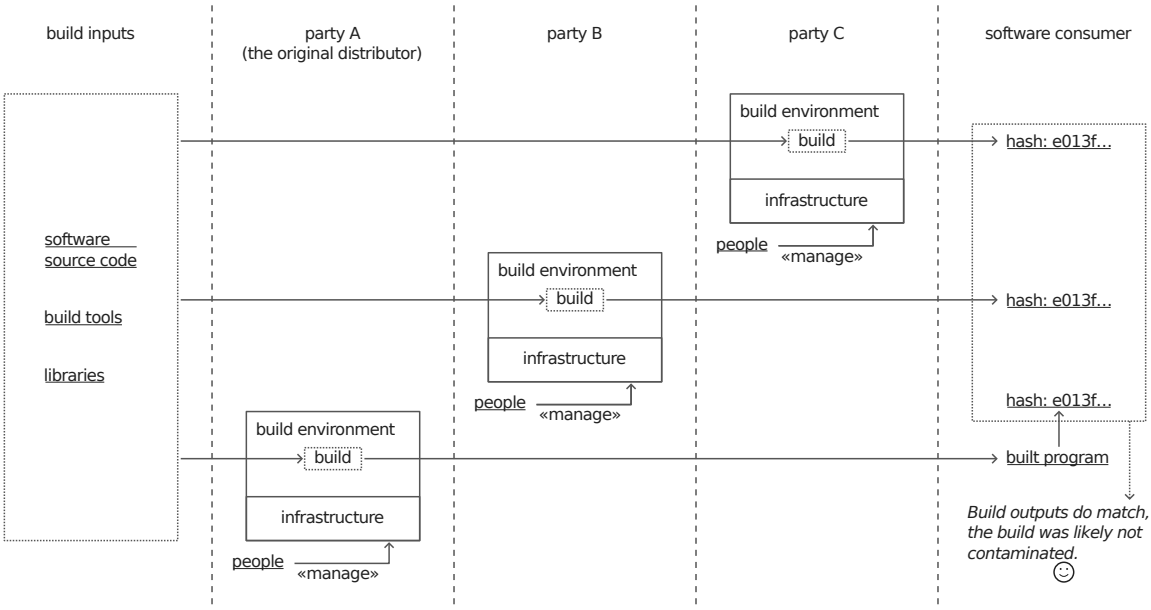

Figure 1.2: Overview of a successful multiparty build verification process.

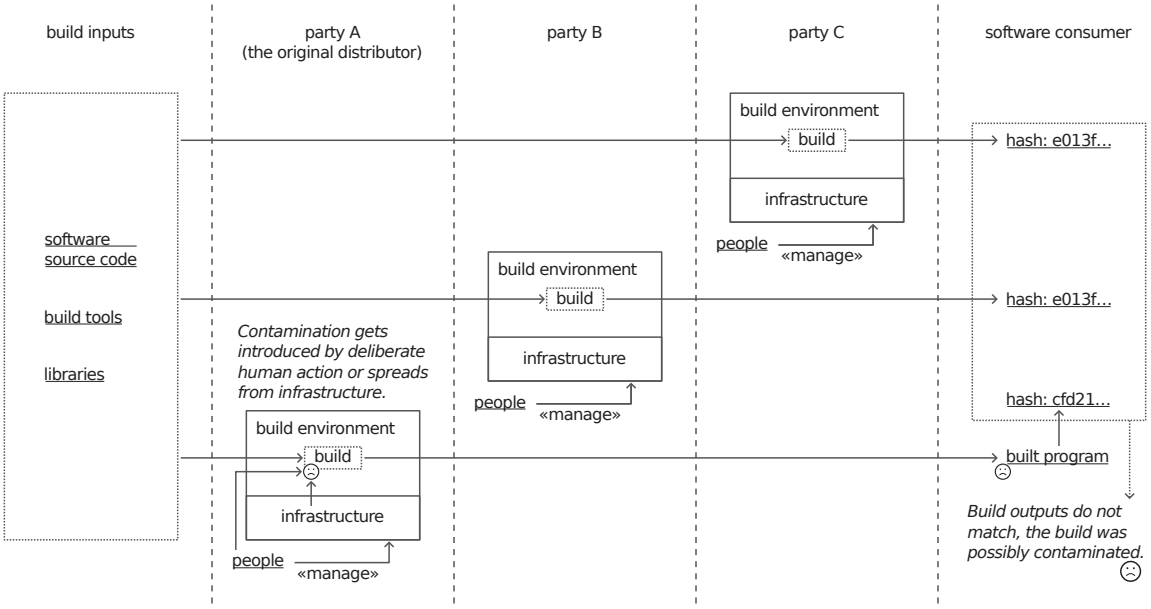

Figure 1.3: Overview of an unsuccessful build verification process.

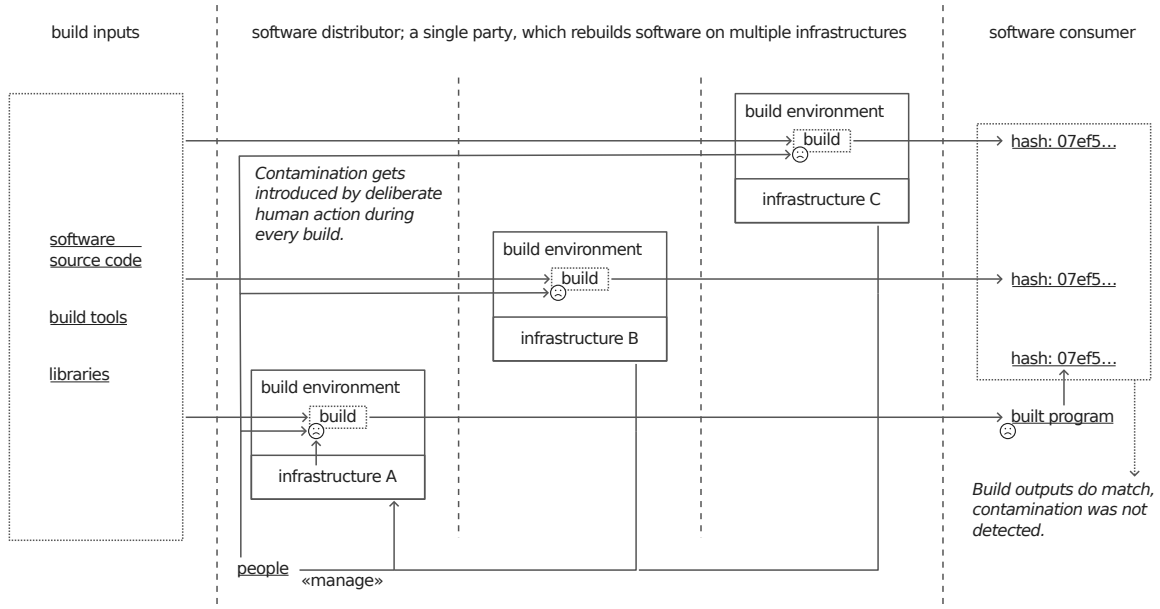

Figure 1.4: Overview of a build verification process which only compared the outputs of builds performed by a single party and failed to detect malice.

quality of reproducibility, where repetition of the same build produces output without variations. Achieving that quality can itself be challenging. This involves identification of sources of build process' nondeterminism – like timestamps and a changing order of filenames in directory listings. Identified sources need to be removed, e.g., by use of fixed date in timestamps or sorting of filenames obtained from directory scans. Achieving this is nowadays easier because common sources of nondeterminism have already been investigated and workarounds have been implemented. For example, since version 7 the GNU C Compiler checks for the existence of a `SOURCE_DATE_EPOCH` environment variable containing a time value. It automatically uses this value in generated file timestamps [Lam+]. Additionally, dedicated tooling for investigating non-reproducibility issues has been developed, notably the **diffoscope** program [LZ21]. To decrease the chance of contamination from a compromized operating system, firmware, and hardware, the actual build – once its reproducibility issues are resolved – should be performed on infrastructures that differ as much as possible, except for the invariant set of build inputs and environment configuration needed to ensure reproducibile outputs.

This work does not address the challenges of avoiding nondeterminism in software builds. Instead, the work's goal is to ensure that – in practical scenarios – the build inputs remain invariant in all build repetition attempts. The work's second major concern is that all machine-performed operations – even those deemed preparatory – can have their

effect on build output controlled through reproducibility. All of this, in turn, can make reproducible builds a more reliable and more complete security mechanism.

Despite their benefits, one should nevertheless realize that reproducible builds only address a particular subset of software supply chain threats – ones affecting the build process – and are mostly useful if other stages of that chain are also secured. Some of the practices that can help with this are

1. making sure the software sources relied upon are audited against backdoors, at least informally, e.g., by the virtue of being developed in the Bazaar model, where the public has a high chance of noticing malicious changes [Ray01],

2. making sure the software sources do not get altered by any party after the verification from the step above, and

3. using reproducible builds and other mechanisms to verify software's dependencies, possibly recursively.

One example of a threat not remediated through reproducible builds alone is the loud XZ backdoor from 2024. Among others, it targeted Debian and Fedora software distributions. Backdoor's activation code was only present in the official source release archives and not in the source repository, which is most often looked at by programmers [Lin+24]. When the software was built from source archives, it had backdoor code linked in, but build results were deterministic. Attacks in such form would not be possible if the source archives were verified to correspond to version-controlled software sources.

2. Contemporary guidance and standards in the field of software supply chain security

Threats related to software build and delivery process were known already long ago. One interesting self-implanting compiler backdoor was described by Ken Thompson in his Turing Award lecture in 1984, "Reflections on Trusting Trust" [Tho84]. Later signs of interest in supply chain threats in certain circles include for example David A. Wheeler's PhD dissertation titled "Fully Countering Trusting Trust through Diverse Double-Compiling" [Whe09] and eventually Bitcoin's use of Gitian deterministic builder[1]. Also, for many years certain software distributions have been compiling their software from sources on dedicated servers, in isolated environments with only mininum dependencies for a given build. One example of such distribution is Debian.

At the same time, for a wide public it's been a norm to rely on prebuilt software that gives virtually no guarantees that it was built in a secure environment. Multiple software repositories allow publishing developer-built software packages. Such software – built from a valid VCS[2] checkout but on developer's infected machine – could be maliciously modified and distributed to unaware software integrators wishing to utilize it in their projects. Neither cryptographic signing of packages nor VCS source code audits would mitigate such attacks.

Several spectacular supply chain attacks of recent years became the catalyst of work towards increasing the level of security. In case of the SolarWinds attack from 2020, also known under the name Sunburst, software distributed among reportedly more than 18 000 customers turned out to contain a backdoor implanted after a compromise of vendor's infrastructure [SB21]. It was exactly the kind of threat that reproducible builds address. As a result of the event, SolarWinds Corporation suffered great economic losses and pejoration of its brand's image. Additionally, the company exposed thousands of

[1] before replacing Gitian with GNU Guix in 2021
[2] version control system

customers to cyberattacks leveraging its compromised software. All of this could have been avoided through reproducible verification of software build outputs.

As more attacks on software build and distribution are reported, software supply chain security becomes a hot topic. It attracts the attention of public institutions and private organizations alike. Some prominent undertakings by nonprofits are: launch of OWASP's[3] SCVS[4] in 2019, foundation of OpenSSF[5] in 2020, launch of its SLSA[6] framework in 2021, Microsoft's donation of S2C2F[7] to OpenSSF in 2022, as well as the publishing of CNCF's[8] "Software Supply Chain Best Practices" in 2021. State actors also took voice by the means of "Securing the Software Supply Chain: Recommended Practices for Developers" and subsequent two guides from 2022 developed by the ESF[9] partnership with support from CISA[10], the NSA[11], and the Office of the Director of National Intelligence. Another possibly relevant document is NSA's "Recommendations for Software Bill of Materials (SBOM) Management" from 2023.

2.1 Software Component Verification Standard

SCVS [Spr+20] describes itself as a "community-driven effort to establish a framework for identifying activities, controls, and best practices, which can help in identifying and reducing risk in a software supply chain". Despite being developed by OWASP it is generic and not limited to web applications in its scope. Authors recognize the unfeasibility of applying all good practices and threat mitigations at every phase of every software project and categorize their security requirements into three levels, each implying the previous one and extending it.

Practices listed in SCVS are grouped into six topics and formulated briefly. They are agnostic about the technology stack and data formats in use. At length explanation of the importance of prescribed actions is not part of the document.

As of version 1.0 of the standard, level 2 requirements include a method to locate

[3]Open Worldwide Application Security Project
[4]Software Component Verification Standard
[5]Open Source Security Foundation
[6]Supply Chain Levels for Software Artifacts
[7]Secure Supply Chain Consumption Framework
[8]Cloud Native Computing Foundation, a project of Linux Foundation
[9]Enduring Security Framework
[10]Cybersecurity and Infrastructure Security Agency
[11]National Security Agency

"specific source codes in version control" that correspond to a given version of a third-party package from software repository. While it is stated that the correspondence must be verifiable, further details are not given. SBOM[12] and repeatable build process are required for an application being developed but not for third-party components. Additionally, listed practices regarding the build environment mention neither the goal of reproducibility nor the weaker hermeticity. While authors might have – justifiably – judged such rules as unfeasible given the current state of the software ecosystem, it is interesting from the point of view of this work. Threats that could not be addressed a few years ago in a generic setting might be remediable now in the context of one or several technology stacks.

2.2 Supply Chain Levels for Software Artifacts

SLSA [Adr+25] uses similar but conceptually more complex categorization than SCVS. Practices are going to be assigned to so-called "tracks" which correspond to different aspects of software supply chain security and which might use different numbers of security levels. As of framwork version 1.0 there only exists a "Build" track with three levels, not counting the empty zeroth level. In addition to the specification of requirements, SLSA documents different threats grouped into those concerning the source, dependencies, build, availability, and verification of an artifact. Historical examples of attacks using some of these techniques are listed in the documentation. Finally, it also includes instructions how to apply SLSA and how to use it with attestation formats from the in-toto framework [Tor+19].

Many aspects of the build process are addressed in the specified requirements set but the qualities of hermeticity and reproducibility were removed from the set at the drafting stage. SLSA explicitly calls verified reproducible builds one of multiple methods of implementing the requirements. In the context of the particular threat of compromised infrastructure, framework's focus is instead on stronger security controls for the build platform. The platform, however, remains a single point of failure. Incidents like that of SolarWinds could still occur. Reproducibility and hermeticity might be re-introduced in subsequent revisions of SLSA, as explained on its "Future directions" page.

The specification currently also does not cover the recursive application of its require-

[12] software bill of materials

ments to input artifacts used. It is nonetheless suggested that users could apply SLSA independently to transitive dependencies. This approach is presented as a possible mitigation to attacks like that performed on event-stream library in 2018.

2.3 Secure Supply Chain Consumption Framework

S2C2F [Dig+22] is complementary to SLSA in that it embraces software consumer's point of view. It introduces four "levels of maturity" of requirements with the highest level mandating a consumer-performed rebuild of all artifacts. Having the artifact built reproducibly by several third parties is mentioned as an alternative approach. Neither method is presented as more secure, even though local rebuild still suffers from being a single point of failure.

2.4 "Software Supply Chain Best Practices"

As of version 1, this paper [Veg+21] recognizes three categories of risk environments and three categories of assurance requirements. These are – in both cases – "low", "moderate", and "high". A methodology for securing the software supply chain is presented in five stages, with four themes of "Verification", "Automation", "Authorization in Controlled Environments", and "Secure Authentication" being repeated in them. Recommendations are organized into paragraphs rather than tables or lists. Authors point towards existing tools useful for some of the tasks, notably the in-toto framwork [Tor+19] and Rebuilderd system [DHV]. At the same time, they openly admit that some of the practices they describe might require extra effort to implement because certain challenges have not yet been countered by the supply chain industry.

Reproducible builds are presented as potentially leverageable when high assurance is needed. The topic is discussed in more detail than in the previous documents from OWASP and OpenSSF. In addition, hermeticity is included as a recommendation for high-risk and high-assurance environments. Recursive dependencies are treated with equal care to the direct ones, consistently with authors' statement that "a supply chain's security is defined by its weakest link". The issue of bootstrapping a system image for builds is also discussed in the paper.

2.5 "Securing the Software Supply Chain: Recommended Practices Guide"

The series was developed by a public-private working group with members from both the industry and U.S. government agencies. It is described as informational only and does not define any standard. Subsequent parts are addressed at software developers, suppliers – who are considered to be "liaising between the customer and software developer" – and customers.

Although these series do not group recommendations into levels, two mitigations in the first guide from August 2022 [NOC22] are called "advanced" and described as providing "additional protection". These are the hermetic and reproducible builds. A suggestion is made that the same builds are performed "in both cloud and on-premise environments" and their outputs compared. Additionally, authors state a justification should be required when it is impossible to perform certain build reproducibly. The text of this requirement has been copied verbatim from SLSA draft back before being removed there.

The guide also recommends that images used to deploy the build environment should be created from sources except where "there is an understanding of the provenance and trust of delivery". No statements explicitly concerning rebuilds of transitive dependencies of a product are made.

2.6 "Recommendations for SBOM Management"

The paper [NSA24] calls itself a guidance. It lists recommendations for general software suppliers and consumers but also dedicates a big part to users and owners of NSS[13]. Document's primary focus is on functionalities that tools used to manage SBOMs should provide.

NSA's guidance concerns SBOMs, which hold information about software components comprising final product. The guidance does not directly address build process threats and does not touch the topics of reproducibility and transitive dependencies of software. In fact, the industry recognizes another type of bill of materials, not mentioned in the

[13]U.S. national security systems – a specific category of information systems used on behalf of U.S. agencies

document, which is more relevant to the topic of reproducibility than SBOM. It is manufacturing bill of materials. In the context of software, MBOM conveys information about all components needed for its build. This also includes project's build dependencies which would not be recorded in an SBOM. MBOMs are relevant from reproducibility perspective because information in them can make software rebuilds possible. Even though MBOMs are not directly mentioned in version 1.1 of NSA's guidance, one of the recommendations present there is labeled as "Scalable architecture". It is described as one that can also "handle other types of BOMs".

2.7 Summary

Published documents' attitutes to reproducibility and hermeticity range from agnosticism to suggestion and recommendation in the context of certain environments. Reproducibility and its requisite – hermeticity – are difficult to achieve with a great subset of existing popular software projects. This difficulty might stand behind the limited focus on these measures in documents other than CNCF's "Software Supply Chain Best Practices". It appears that the means of securing the software supply chain which are more straightforward to employ are also more often recommended. In such case, making reproducibility easier to achieve for all kinds of software projects should lead to it being more frequently discussed and therefore more broadly leveraged.

3. Security tools leveraging reproducible builds

Several initiatives and pieces of software exist that are concerned with the verification of reproducibility of software packages. The champion of these efforts is the Reproducible Builds project, also affiliated with Debian.

3.1 in-toto apt-transport for Debian packages

in-toto framework, developed under the CNCF, aims to secure the integrity of software supply chains [Tor+19]. Debian GNU/Linux is an operating system distribution founded in 1993. It provides thousands of pieces of software in form of **packages** that can be installed in the system via Debian's package manager, APT[1].

In 2018 it became possible to use in-toto together with Debian's APT, to verify that a package being installed has been verified through reproducible builds. The package manager can be configured to abort installation if the package was not reproduced by at least k independent rebuilders, with k configurable by the user. In the process, cryptographically signed attestations of rebuilt packages are fetched over the network from rebuilder URIs that are also configured by the user.

3.1.1 How a Debian package is made and rebuilt

Most packages available in Debian contain software written by third parties, i.e., the **upstream**. That software was released in source form and subsequently integrated into Debian. A package typically has a maintainer who is a Debian volunteer taking care of the package, possibly with the aid of other people [Rod+22, Chapter 1].

Upon initial creation or a version update of a Debian package, its maintainer first prepares what is called a **source package**. It is the primary input of the build process

[1] Advanced Package Tool

that will produce the final, installable package. The installable package is also sometimes called a **binary package** to distinguish it from the source package. A single build process with a single source package can also prouce multiple binary packages. For example, a programming library written in the C programming language can have its dynamically linked binary and its header files placed in distinct binary packages. As of June 3rd, 2025, the current release of Debian – Debian 12, codenamed "Bookworm" – offered 63 465 packages for x86_64 architecture, as found in its `main` pool. They were produced from 34 217 source packages.

An official Debian source package typically consists of

1. software sources taken from the upstream – possibly with inappropriately licensed components removed and other changes applied to meet Debian guidelines – taking form of one or more compressed archives,
2. package's recipe, taking form of a compressed archive, including, among others,
   - a list of software's build and runtime dependencies, placed – with other metadata – in a file named `debian-control`, with an example in Listing 3.1,
   - optional patch files that describe Debian-specific changes which are to be applied to upstream software as part of the automated build process, with an example in Listing 3.2, and
   - a script directing the build process, placed in a file named `debian/rules`, invoked as a Makefile, with an example in Listing 3.3, and
3. a text file with `.dsc` suffix containing cryptographically signed source package metadata, including hashes of compressed archives from the above points.

The package maintainer is likely to perform one or several builds when working on a new or updated source package. However, except for special cases, it is only the source package and not the maintainer-built binary packages that gets uploaded to what is called the **Debian archive**. Uploaded source packages are "built automatically by the build daemons in a controlled and predictable environment" [Lev+25, Chapter 5].

Besides producing binary packages, the build daemons also record the metadata of performed builds, which is later published with cryptographic signatures as `.buildinfo` files[2].

[2]which can be considered a type of MBOM that was described in 2.6

```
Source: nodejs
Section: javascript
Priority: optional
Maintainer: Debian Javascript Maintainers <pkg-javascript-devel@alioth-lists.debian.net>
Uploaders: Jérémy Lal <kapouer@melix.org>,
 Jonas Smedegaard <dr@jones.dk>
Build-Depends:
 sse2-support [i386] <!nocheck>,
 armv6k-support [armel] <!nocheck>, vfpv2-support [armel] <!nocheck>,
 debhelper-compat (= 13),
 dh-buildinfo,
 bash-completion,
 dh-sequence-bash-completion,
 ca-certificates,
 curl <!nocheck>,
 gyp (>= 0.16.0~),
 jq,
 libbrotli-dev,
[...]

Package: nodejs
Architecture: amd64 arm64 armel armhf i386 mips64el mips64r6el loong64 powerpc ppc64 ppc64el riscv64 s390x
Multi-Arch: allowed
Depends:
 ${shlibs:Depends},
 ${misc:Depends},
 node-corepack <!pkg.nodejs.nobuiltin>,
 sse2-support [i386],
 armv6k-support [armel], vfpv2-support [armel],
 libnode115 (= ${binary:Version})
Recommends: ca-certificates,
 nodejs-doc
Suggests: npm
[...]
```

Listing 3.1: Excerpts from a 176-lines long `debian/control` file of the `nodejs` Debian package.

```
1  Description: fix rc path to allow build system run tests
2  Author: Joao Eriberto Mota Filho <eriberto@debian.org>
3  Last-Update: 2019-10-20
4  --- shc-4.0.3.orig/test/ttest.sh
5  +++ shc-4.0.3/test/ttest.sh
6  @@ -1,6 +1,6 @@
7   #!/bin/bash
8
9  -shells=('/bin/sh' '/bin/dash' '/bin/bash' '/bin/ash' '/bin/ksh' '/bin/zsh' '/usr/bin/tcsh' '/bin/csh' '/usr
   ↪ /bin/rc')
10 +shells=('/bin/sh' '/bin/dash' '/bin/bash' '/bin/ash' '/bin/ksh' '/bin/zsh' '/usr/bin/tcsh' '/bin/csh' '/bin
   ↪ /rc')
11  ## Install: sudo apt install dash bash ash ksh zsh tcsh csh rc
12
13  check_opts='' '-r' '-v' '-D' '-S')
```

Listing 3.2: A patch used by Debian package `shc` to provide an upstream script with the correct path of the `rc` executable, as present in Debian.

```
[...]

stamps/build.%: stamps/configure.%
        $(checkdir)
        @echo BEGIN $@
        env BFD_SOVER_EXT="-$*" CTF_SOVER_EXT="-$*" \
          $(call SET_BINUTILS_MULTIARCH_ENV,$*) \
          $(MAKE) -C builddir-$* $(NJOBS) \
                CFLAGS="$(CFLAGS)" \
                CXXFLAGS="$(CXXFLAGS)" \
                LDFLAGS="$(LDFLAGS) -Wl,-z,relro"
ifeq ($(DEB_BUILD_GNU_TYPE),$(DEB_HOST_GNU_TYPE))
ifeq ($(with_check),yes)
        -env MAKE="$(MAKE) VERSION=$(VERSION)-$*" \
          $(call SET_BINUTILS_MULTIARCH_ENV,$*) \
            $(MAKE) -C builddir-$* -k check
        rm -f $(pwd)/test-summary-$*
        for f in \
          builddir-$*/binutils/binutils.sum \
          builddir-$*/gas/testsuite/gas.sum \
          builddir-$*/ld/ld.sum \
          builddir-$*/libctf/libctf.sum \
          builddir-$*/gprofng/gprofng.sum \
          builddir-$*/libsframe/libsframe.sum \
        ; do \
[...]
```

Listing 3.3: Excerpt from a 2045-lines long `debian/rules` Makefile of the `binutils` Debian package.

For a given binary package, it is usually possible to locate and download its corresponding `.buildinfo` file. That file contains, among others, a list of names and versions of Debian packages that were installed in the minimal build environment. An example of a `.buildinfo` file is shown in Listing 3.4.

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA512

Format: 1.0
Source: haskell-base-compat-batteries
Binary: libghc-base-compat-batteries-dev libghc-base-compat-batteries-prof
Architecture: amd64
Version: 0.11.2-1
[...]
Checksums-Sha256:
 db28e3a63a2306507c471693da3cd9a06609aabe8189712e7759ad7919e60b1f 82332 libghc-base-comp
at-batteries-dev_0.11.2-1_amd64.deb
 f1a76638f3ce5f5584241363da07e01582b72837ade73ba0bdacf3bb221b366a 63004 libghc-base-comp
at-batteries-prof_0.11.2-1_amd64.deb
Build-Origin: Debian
Build-Architecture: amd64
Build-Date: Wed, 15 Jun 2022 00:46:44 +0000
Build-Path: /build/haskell-base-compat-batteries-rkofMw/haskell-base-compat-batteries-0.
11.2
Installed-Build-Depends:
 autoconf (= 2.71-2),
 automake (= 1:1.16.5-1.3),
 autopoint (= 0.21-6),
 autotools-dev (= 20220109.1),
 base-files (= 12.2),
[...]
```

Listing 3.4: Excerpt from a 256-lines long `.buildinfo` file of the `haskell-base-compat-batteries` Debian package.

The `.buildinfo` files can be used by parties other that The Debian Project to rebuild the official packages, write in-toto metadata of the process, sign it, and subsequently serve it to the users.

3.2 `guix challenge` command of GNU Guix

GNU Guix is an operating system distribution and a package manager that appeared in 2013 [Cou13]. It implements functional package management model described by Eelco Dolstra in "The Purely Functional Software Deployment Model" in 2006 [Dol06] and pioneered by Nix package manager. For avoidance of confusion with an unrelated "GUIX" U.S. trademark registered in 2019 and owned by Microsoft, the GNU Guix package manager shall be referred to with its "GNU" prefix throughout this document.

Similarly to Debian, GNU Guix relies on software written by upstream authors and makes it available in the form of installable packages. However, the data formats, build mechanisms, and nomenclature differ. The equivalent of Debian's binary package is referred to as a **substitute**. Since 2015, GNU Guix provides a `guix challenge` command which "allows users to challenge the authenticity of substitutes provided by a server" [Cou15]. End users can invoke this command to compare the outputs of package builds performed on multiple infrastructures and report which packages were not built reproducibly – either due to nondeterminism of the build process or because of build's compromise.

3.2.1 How a GNU Guix package is made and built

GNU Guix package collection is determined by a set of package recipes. Unlike Debian package recipes, these do not take the form of compressed achives. Here, packages are defined in Scheme programming language from Lisp family. A recipe consists of code that instantiates and populates a `<package>` data structure representing a package that can be built. Recipe's code can use the facilities of the Turing-complete Scheme programming language to dynamically compute parts of this new `<package>` instance. A `<package>` instance does – in most cases – get bound to a Scheme variable, which other recipes' code can reference. Lists of package's explicit build and runtime dependencies are typically constructed using references to other package variables. A package recipe example is shown in Listing 3.5. It defines a variable of the same name as the package and declares its

explicit dependencies by referencing `python-protobuf` and other two package variables. A Scheme code snippet is supplied to be executed during the hermetic build process. It customizes the process by deleting unnecessary files.

```
(define-public python-axolotl
  (package
    (name "python-axolotl")
    (version "0.2.3")
    (source
     (origin
       (method url-fetch)
       (uri (pypi-uri "python-axolotl" version))
       (sha256
        (base32
         "1bwdp24fmriffwx91aigs9k162albb51iskp23nc939z893q23py"))))
    (build-system python-build-system)
    (arguments
     `(#:phases
       (modify-phases %standard-phases
         ;; Don't install tests
         (add-before 'install 'remove-tests
           (lambda _
             (for-each delete-file-recursively
                       '("axolotl/tests" "build/lib/axolotl/tests"))
             #t)))))
    (propagated-inputs
     (list python-axolotl-curve25519 python-cryptography python-protobuf))
    (home-page "https://github.com/tgalal/python-axolotl")
    (synopsis "Python port of libaxolotl-android")
    (description "This is a python port of libaxolotl-android.  This
is a ratcheting forward secrecy protocol that works in synchronous and
asynchronous messaging environments.")
    (license license:gpl3)))
```

Listing 3.5: Recipe of `python-axolotl` GNU Guix package.

The recipes of all official GNU Guix packages are kept and maintained in a single VCS repository. As of April 14th, 2025, this is a repository that houses both the recipes collection and the GNU Guix application, although this setup is not imposed by design and might change in the future. Recipes in the repository can also have accompanying patch files. However, patches are used less frequently here than in Debian package recipes. Regular expression substitutions performed by Scheme code are preferred by GNU Guix developers for trivial modifications of upstream source.

Package recipes in GNU Guix generally reference remote software sources using URLs and hashes that are considered cryptographically secure. The hashes are used for verification of sources' integrity upon download and make it possible to safely download them from fallback servers, among which is the archive of Software Heritage [Cou+24]. Several remote resource formats are supported, including traditional compressed archives as well as repositories of popular VCSes. Referencing VSC repositories of upstream projects – although not practiced universally in the recipes collection – allows the correspondence of build inputs to public-reviewed sources to be more easily tracked.

The GNU Guix package manager is able to build packages locally, on the system on which their installation was requested. Each deployment of GNU Guix comes – by design – with a copy of the recipes collection, such that only the source inputs – identified by cryptographic hashes – need to be downloaded for a build to be performed. Local builds are fully automated, are performed in isolated environments created in the background and are a first-class citizen in the model of GNU Guix. For practical reasons, it is made possible to instead download prebuilt packages – the substitutes that substitute their locally-built equivalents.

The `guix challenge` command allows the build outputs advertised by configured substitute servers to be compared with each other and with the outputs of local builds, when available.

Lack of automatized integration of reproducibility verification with package deployment is a notable limitation of the `guix challenge` command of GNU Guix. The command has to be invoked explicitly by the user. As of April 14th, 2025, there is no official way to **automatically** challenge the binary substitutes that GNU Guix downloads as part of other actions, such as software installation. Thus, in practice, the command is less easily usable as an end user's preventive security tool and more as an investigation and internal verification aid.

Bitcoin Core, possibly the most famous piece of software using GNU Guix to reproducibly verify its binaries, does not rely on the `guix challenge` command and instead uses its own custom scripts to perform code signing.

3.3 Continuous tests

The tools presented so far allow the end users to verify binaries before they are put in use. The user can first learn whether a set of software packages was rebuilt with bit-to-bit identical results on independent infrastructure and can then make an informed decision whether to install the packages. The benefit of this type of verification is that it leaves no single point of failure, except for end user's device. However, if the latter were compromised in the first place, no software-based scheme would reliably remediate that. This scenario is therefore out of scope of this work.

The drawback of this type of verification is that accidential non-reproducibility due to

an overlooked source of nondeterminism in the build process leads to verification failures, as depicted in Figure 3.1. A lag of independent rebuilders can likewise make verification impossible, as also shown in the figure. If reproducible builds are to be used as a preventive security measure, any such failure would need to stop the end user from performing the attempted software installation or update. Until the percentage of reproducibly buildable software in distributions is close to 100% and enough resources are invested in independent infrastructure performing continuous rebuilding, this problem can be prohibitive.

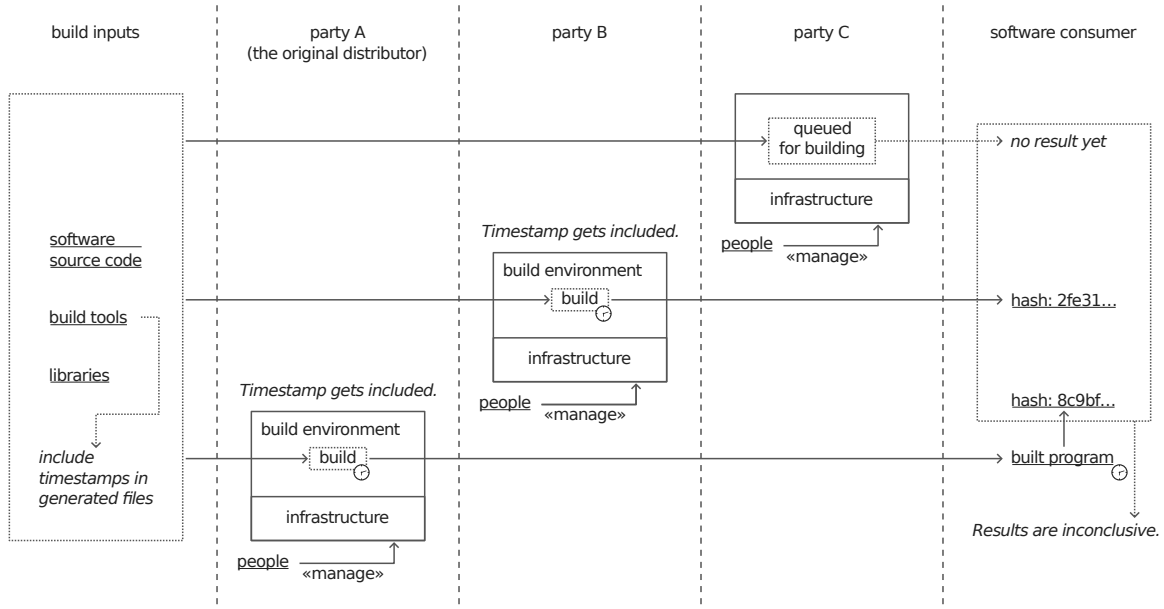

Figure 3.1: Overview of a verification process inconclusive due to rebuilder's delay and lack of determinism.

However, there are several projects where verification of reproducibility is performed by someone else than the end user. Although this does not eliminate the single point of failure from software installation process, such verification can still make supply chain attacks harder. For example, if an organization performs internal tests of reproducibility and analyzes their results, it is more likely to detect code contamination early on and react to it.

As of June 3rd, 2025, the Reproducible Builds project performs continuous tests of the reproducibility of

- files from coreboot, FreeBSD, NetBSD, and OpenWrt projects, as well as
- packages from Debian repositories, with package reproducibility statistics being reported, as in Figure 3.2.

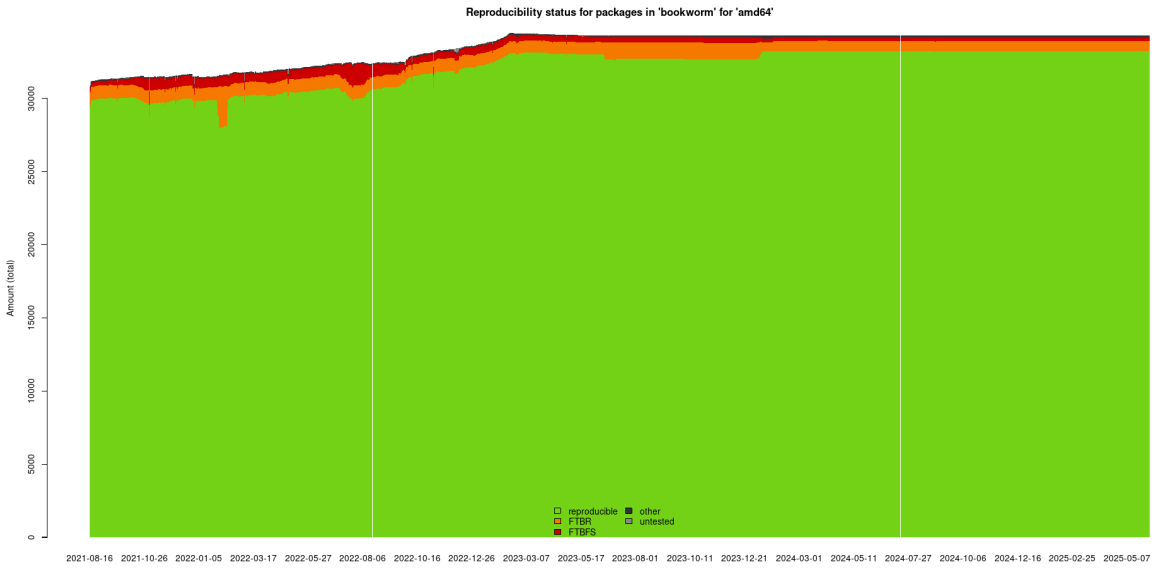

Figure 3.2: Reproducibility of Debian Bookworm packages over time, as presented on Reproducible Builds' continuous tests website.

As of June 3rd, 2025, 33 214 source packages from Debian Bookworm were reported to have been rebuilt reproducibly for the x86_64 architecture. That means approximately 97% reproducibility in the collection. The remaining packages either could not be built on the Reproducible Builds infrastructure, with various possible reasons, or were built with outputs differing on binary level.

The Reproducible Builds project also lists several others – including GNU Guix mentioned earlier and its predecessor NixOS – that monitor the reproducibility of their files and/or repository packages without relying on the Reproducible Builds' infrastructure [RBCT]. One notable undertaking in this category is the development of Rebuilderd tool for reproducible rebuilds of packages from Arch Linux and recently other distributions [DHV]. An application also exists that can consult a Rebuilderd instance to automatically verify packages installed in user's Arch Linux system [Lkcnt].

The continuous tests platform used by GNU Guix is capable of generating reproducibility reports, which are viewable on pages at `https://data.guix.gnu.org`. Part of such report is shown in Figure 3.3. According to it, there were 39 344 packages available for the x86_64 architecture as of April 14th, 2025, GNU Guix Git commit `143faecec3`. 35 415 of them – approximately 90% – were rebuilt reproducibly, albeit with 2 087 remaining untested. Frequent package updates and builders' lag are possible reasons for the large number of untested packages. Out of all the successfully rebuilt GNU Guix packages,

approximately 95% had outputs that were bit-to-bit identical with those produced on another infrastructure.

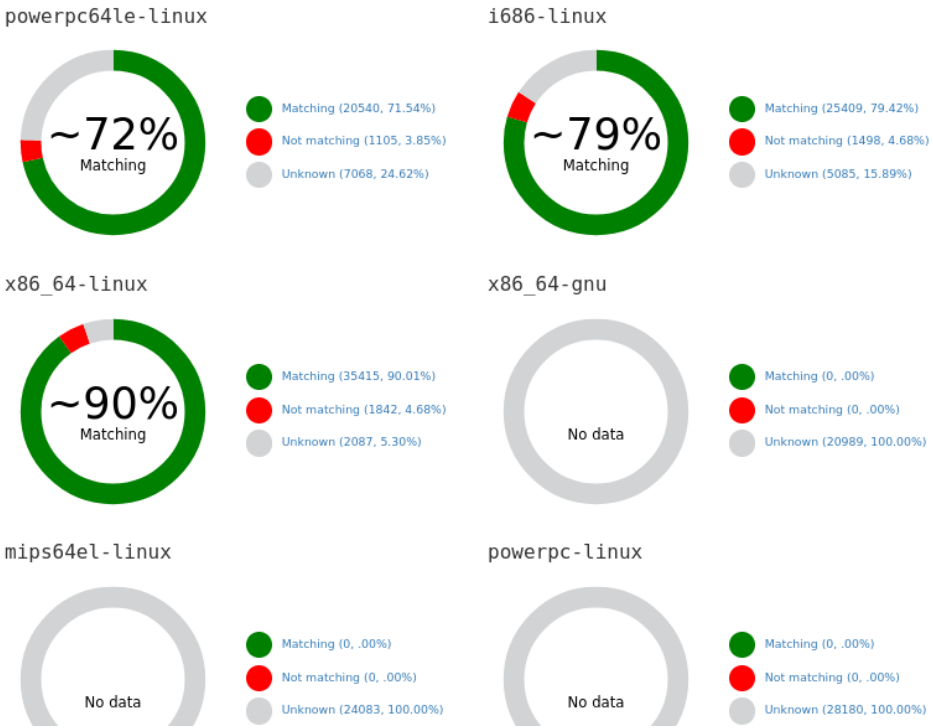

Figure 3.3: Reproducibility of GNU Guix packages as reported by its continuous tests platform.

Unfortunately, several of the reproducibility tests listed on the Reproducible Builds website have become unmaintained. The testing for Fedora and Alpine operating system distributions was disabled at some point. Although the reproducibility statistics of GNU Guix packages are still delivered, their web pages sometimes cannot be viewed due to timeouts, as also witnessed by Internet Archive's Wayback Machine[3].

[3] https://web.archive.org/web/20250625124729/https://data.guix.gnu.org/repository/1/branch/master/latest-processed-revision/package-reproducibility

4. Applicability of reproducibility workflows to different software ecosystems

Current reproducible software distributions, like GNU Guix and Debian, are **system software distributions** – ones that contain a collection of packages that can form a complete operating system. As such, a mixture of software technologies can be found in them.

Certain programming languages and computing platforms form software ecosystems centered around them, for instance, the ecosystem of the Python programming language with CPython[1] and PyPy being its most popular runtimes. These ecosystems evolve their specific software package formats and workflows for building these packages. Many popular ecosystems have their dedicated package repositories that usually serve as primary distribution channels of software written for the ecosystem's computing platform. Such ecosystem-specific software repositories are often, although imprecisely, referred to as language-specific repositories. They are typically open to the public for registration and package creation. As such, they form environments of software with varying levels of quality and significance.

Many ecosystem-specific repositories distribute software without all the metadata that is necessary to automate rebuilding it. Moreover, if a project uses multiple packages from such repository, it relies on security of each of the machines used by these packages' developers for builds and uploads. A partial remedy – facility to publish packages together with build provenance data cryptographically signed by a build service – was employed by ecosystem-specific repositories **npm Registry** and **PyPI**[2] in 2023 and 2024, respectively. A dedicated build service – with the most popular ones being GitHub Actions and GitLab CI/CD – can be considered better secured than an average developer's computer, for the benefit of packages that are confirmed to have been built there. In addition, build

[1] the most popular implementation of the Python programming language, written in C
[2] Python Package Index

provenance data identifies the source repository used in the build. However, even when secured, build service remains a single point of failure of the process. In addition, support for only one or few selected build services – as offered by the npm Registry as of April 14th, 2025 – leads to vendor lock-in.

4.1 Degree of inclusion in Debian and GNU Guix

Software utilizing the respective computing platforms and distributed primarily through an ecosystem-specific software repository might, at some point, also get included in a system software distribution. However, so far it did not happen with certain popular and strategic pieces of software. One example is Electron framework that is used, among others, by Signal application and Visual Studio Code IDEs. As of April 14th, 2025, Electron is declared a development dependency by 4533 packages in the npm Registry. At the same time, software distributions that test for package reproducibility usually lack Electron and Electron-based applications, as do Debian and GNU Guix. Another software distribution that tests for package reproducibility, NixOS, redistributes Electron's upstream binaries without actually building the software. In this case, the build itself is not being verified through reproducibility.

Certain programming languages and computing platforms have seen more packaging progress in system software distributions. Let us consider the PyPI, npm, and crates ecosystems, which are centered around their respective repositories and technologies, as shown in Table 4.1. For this work, repositories were chosen as a basis for distinguishing the ecosystems. It is a feasible criterion, although not the only possible one. There are overlaps of various sizes between different repositories, runtimes, and project management tools. Also, in some cases a software package has formal of informal dependencies that are not distributed through the reposotory that the package itself uses.

We shall compare the numbers of software projects from the chosen ecosystems that are packaged in system software distributions described in detail in 3. The numbers presented in Table 4.2 were estimated based on snapshots of package collections offered by Debian Bookworm as of June 3rd, 2025 and GNU Guix as of April 14th, 2025, GNU Guix Git commit `143faecec3`.

A conclusion arises that for some reason npm packages are less likely to be packaged

Table 4.1: Considered software ecosystems.

	PyPI	npm	crates
repository	PyPI	npm Registry	crates.io
primary programming languages	• Python, • Cython	• JavaScript, • TypeScript, • WASM	• Rust
sample runtimes or compilers	• CPython, • PyPy	• Node.js, • Deno, • Bun	• rustc
sample project management tools	• setuptools, • Poetry, • Hatch	• npm, • Yarn, • pnpm	• Cargo

Table 4.2: Estimated numbers of Debian and GNU Guix packages corresponding to software from considered ecosystems.

	PyPI	npm	crates
GNU Guix packages	3699	55	3704
estimated as use counts of which `build-system`s in recipes	`pyproject-build-system`, `python-build-system`	`node-build-system`	`cargo-build-system`
Debian packages	5312	998	1424
estimated as counts of source packages referencing which debhelper package	`dh-python`	`dh-nodejs`	`dh-cargo`

when adhering to the rigor of existing software distributions that utilize hermetic and reproducible builds. We can try to name the main causes and judge whether the difficulties could be worked around without sacrificing security.

4.2 Dependency tree sizes

It can be noticed that on avarage, npm projects have more recursive dependencies than, for example, Python projects [Boj20]. This means that packaging an end-user application written in JavaScript[3] typically requires more labor of bringing the intermediate packages to the distribution – an issue that has been talked about in the GNU Guix community for at least ten years [Lem15].

Large dependency trees can be partially caused by the JavaScript language historically having a relatively modest standard library. While such design can bring some benefits,

[3]also referred to by its official name: ECMAScript

it might also lead to proliferation of small libraries that have overlapping functionality. Independent, competing packages with similar purposes are then more likely to appear together in a single dependency tree.

Creation of many small packages and eager use of dependencies for simple tasks – all of which leads to larger dependency trees – can also be attributed to the culture of developers working with the npm Registry [Abd+20].

4.3 Age of the ecosystem

The npm tool first appeared in 2010. The PyPI ecosystem is older, with its repository being launched in 2002. It can therefore be argued that software from the latter has had more time to be included in Debian and several other software distributions. However, this is not sufficient to explain the lack of inclusion of npm packages in GNU Guix, which itself came to existence in 2012. Additionally, the crates ecosystem, which came to existence in 2014, is younger than all of the previous repositories. Despite that, software from it has larger presence in Debian an GNU Guix than software from the npm ecosystem.

4.4 Conflicting dependencies

System software distributions typically only allow a single version of a package to be installed at any given time. This rule is sometimes relaxed in various ways. For example, as of June 3rd, 2025, Debian Bookworm had distinct packages named `gcc-12` and `gcc-11`. Both of them provide the GNU C Compiler, albeit in different major versions. These packages can be installed side-by-side. GNU Guix, on the other hand, has facilities to create independent environments with different sets of packages in each. If multiple versions of the same package reside in different environments, they do not cause a conflict. There are also other nuances that provide some degree of flexibility.

Nonetheless, an application that requires multiple versions of a single dependency is more difficult to include in such software distributions. This is a relative small issue for, e.g., Python applications. Their runtime does not support simultaneous loading of multiple versions of the same Python library in the first place. I.e., if it is possible to install package's dependencies from PyPI and use that package, it means there are no

conflicting dependencies. At the same time, npm and the Node.js runtime allow multiple versions of the same library to appear in the dependency tree of a project.

4.4.1 Support in npm&Node.js

Let us consider the dependency tree recorded in `package-lock.json` file of the sigstore project repository[4]. We shall look at the revision designated by Git commit `759e4d9f70` from Aug 5, 2024. Entries of interest are shown in Listing 4.1. A library identified as `tslib` appears two times in the tree. There is a copy of version 2.6.3 and a copy of version 1.14.1. This happened because a dependency, `tsyringe`, has a requirement on a version of `tslib` that is at least 1.9.3 but lower than 2.0.0. Version 1.14.1 present in the tree satisfies this requirement. Another dependency, `pvtsutils`, requires `tslib` in version that is at least 2.6.1 but lower than 3.0.0. Several other entries, omitted for clarity, have a different requirement on `tslib`. All these are satisfied by version 2.6.3.

```
[...]
    "node_modules/pvtsutils": {
      "version": "1.3.5",
      "license": "MIT",
      "dependencies": {
        "tslib": "^2.6.1"
      }
    },
[...]
    "node_modules/tslib": {
      "version": "2.6.3",
      "resolved": "https://registry.npmjs.org/tslib/-/tslib-2.6.3.tgz",
      "integrity": "sha512-xNvxJEOUiWPGhUuUdQgAJPKOOJfGnIyKySOc09XkKsgdUV/3
↪ E2zvwZYdejjmRgPCgcym1juLH3226yA7sEFJKQ=="
    },
    "node_modules/tsyringe": {
      "version": "4.8.0",
      "license": "MIT",
      "dependencies": {
        "tslib": "^1.9.3"
      },
      "engines": {
        "node": ">=␣6.0.0"
      }
    },
    "node_modules/tsyringe/node_modules/tslib": {
      "version": "1.14.1",
      "license": "0BSD"
    },
[...]
```

Listing 4.1: Multiple occurances of `tslib` package in a dependency tree.

When this project's code is run and the `tsyringe` library tries to load `tslib`, Node.js runtime instantiates version 1.14.1 of `tslib` from `tsyringe`'s private subtree. `tsyringe` then

[4] https://raw.githubusercontent.com/sigstore/sigstore-js/759e4d9f706aa0bea883267009fa1da8f2705eab/package-lock.json

works with that instance of `tslib`. For all other parts of the project that attempt to load `tslib`, version 2.6.3 is instantiated and used.

4.4.2 Effects of Semantic Versioning

In the npm ecosystem a system called "Semantic Versioning" [Pre13] is widely applied. This system assumes that software version is given as three numbers – major, minor, and patch. E.g., 2.6.3. It also permits optional labels for pre-release and build metadata. When Semantic Versioning is followed, then a new software release that breaks backward compatibility with the previous release has its major version number increased and the other two numbers reset to zero. A release that adds new functionality without breaking backward compatibility has only its minor number increased, with patch number reset to zero. And a release that adds no new functionality – typically a bugfix release – has its patch number increased.

Assume project `foo` utilizes a semantically versioned dependency `bar`. The developers could verify that `foo`'s code integrates properly with a particular version of `bar`, e.g., version 3.4.5. The developers would then record a requirement that in the future, either this version of `bar` or a later one – but without compatibility-breaking changes – can be used by `foo`. This means only `bar` versions between 3.4.5 and 4.0.0, excluding 4.0.0 itself, would satisfy the requirement. If `bar` then increases its major number in a new release, the developers of `foo` can ensure that its code integrates properly with the newer `bar`. They would then update the requirements in `foo` and the next release of `foo` could officially use the 4.x.x series of `bar`. It would, however, still be forbidden from using the hypothetical 5.x.x series that could bring subsequent compatibility breakages.

In our example from 4.4.1, the libraries `tsyringe` and `pvtsutils` both apply this approach to their dependency, `tslib`. As a result, `tsyringe` is protected from possible breaking changes introduced by version 2.0.0 of `tslib`, but at the same time it is impossible to satisfy all requirements of the project with just a single copy of `tslib`. In practice, there are sometimes tens or hundreds such conflicts in a single dependency tree.

The breaking changes that necessitate increasing package's major version number sometimes concern a part of package's functionality that the specific user does not rely upon. When project's developers know or suspect that the requirements specified by certain dependencies could be safely loosened, they can forcibly override them. Such

overrides, supported natively in npm, are used by some when addressing security vulnerabilities deep in a project's dependency tree. The same approach could be used to eliminate all dependency conflicts. However, with many overrides there's a lower chance of avoiding a breakage due to inter-package compatibility issues.

4.5 Difficult bootstrappability

GNU Guix and Debian are self-contained in the sense that build dependencies of their packages are also their packages. When packaging a program that requires, e.g., a C compiler to build, no problems arise – C compilers are already present in these system software distributions and one of them can be used as a build dependency of the new package. However, packaging a program written in a new programming language requires a compiler or interpreter of that programming language to be present in the distribution in the first place. The same applies to other types of build tools, e.g., bundlers that amalgamate many JavaScript files into a few or a single file.

Packaging a program for such distribution involves first packaging all its build tools. Making a package buildable with only the tools from the distribution is sometimes referred to as **bootstrapping**.

4.5.1 Self-depending software

Certain tools exist that depend on themselves to build, making bootstrapping challenging. Selected examples from the npm ecosystem are presented in Table 4.3. Packages in the table were ranked based on how many other npm Registry packages specified them as development dependencies as of April 14th, 2025. The presented selection is by no means exhaustive, more highly popular self-depending npm packages might exist.

In GNU Guix, the preferred approach to packaging a self-depending tool is making it bootstrappable [Cou22]. This can happen by packaging a chain of historical versions of the tool, where each can be built with the nearest older packaged one, down to an early version that did not have a self-dependency. Sometimes it is possible to eliminate or shorten such "bootstrap chain", for example by replacing a complex build tool with scripts or by using a bootstrappable drop-in replacement to some tool. The latter was an approach used to package the official, self-hosting implementation of the Rust programming language for

Table 4.3: npm Registry packages that require themselves to build.

name	popularity ranking	notes
`typescript`	1 (473235 dependees)	the original implementation of TypeScript programming language
`@babel/core`	10 (138704 dependees)	part of a JavaScript compiler, requring itself indirectly through dependencies that themselves build with `@babel/core`
`rollup`	26 (95965 dependees)	a bundler
`gulp`	40 (61077 dependees)	a build system, requiring itself through its runtime dependency `gulp-cli`
`sucrase`	1793 (528 dependees)	an alternative to Babel, used as a proof of concept for bootstrapping a GNU Guix package

GNU Guix in 2018. There, an unofficial Rust compiler, written in C++, was used to compile an official Rust release from July 2017 [Mil18].

Bootstrapping helps prevent the "Trusting Trust" attack demonstrated by Ken Thompson in 1984, but as of today there is little evidence of such attack type ever being used by threat actors. In some cases software distributions under consideration make exceptions and allow a non-bootstrappable program prebuilt by another party to be made into a distribution package. For example, the set of OCaml and Haskell compilers in GNU Guix depends on such third party binaries that cannot be rebuilt from any package recipe in the distribution.

In 2022 a proof of concept GNU Guix bootstrap of `sucrase`, a self-depending build tool from the npm ecosystem, was done [zam22].

4.5.2 Recursive dependency closure

By package's recursive development dependency closure we mean a set containing all its declared runtime dependencies and development dependencies, their runtime dependencies and development dependencies, etc. In other words, the closure is the minimal set that contains the dependencies of its every member and also of the package for which the closure is being computed. The size of package's recursive dependency closure can illustrate the bootstrapping challenge complexity. An attempt to compute such closure was made for npm package `typescript` as part of this work. npm Registry metadata limited to package releases from before April 14th, 2025 was used. For simplicity, version constraints were disregarded and packages' all historical dependencies were considered. The

result was a 60 843-elements big set of package names, with additional 2 433 referenced names that do not exists in the Registry. These were largely the results of mistakes and possibly private/unpublished packages. Of course, non-crucial tools like linters tend to be declared developments dependencies and the closure of truly necessary dependencies would be much smaller, as also reported in 8.4. Nonetheless, this example shows how difficult it is to reason about what is needed for bootstrapping tasks.

4.6 Inconvenience of system software distributions

Despite looser security practices and more frequent reports of malicious packages in repositories like the npm Registry [Nic22; Mun23; Nap25; Tou25; Lak25], many developers still prefer to work with them rather than with system software distributions. Packages in the latter are adjusted to work with and inside their distributions and are typically not compatible with the usual workflow of developers of, e.g., npm projects. For example, it could be unstraightforward to use npm libraries from Debian for producing distribution files of a mobile application. Another deterrent is the delay with which newer releases of packages reach system software distributions.

This limited interest in availability of software from certain ecosystems in distributions like Debian and GNU Guix also leads to decreased incentive for authors of these distributions to work on it.

5. Overview of the npm ecosystem

Software from the npm ecosystem was found to be more challenging to be made into system software distribution packages. To provide deeper insight into this problem, this chapter provides more information about this ecosystem, with focus on dependency relations between npm packages.

npm Registry – the software repository around which the npm ecosystem is centered – was created as a distribution channel for JavaScript libraries, frameworks, and applications using Node.js runtime. It was targeted towards server-side developers. The platform allows the general public to register accounts and publish software packages. Throughout the years, the npm Registry also became home to client-side JavaScript, i.e., software to be executed in web browsers. The repository is nowadays also used by related programming languages, notably TypeScript. As of April 14th, 2025, the repository was serving over 3.5 million published packages, many of which come in multiple versions.

5.1 Recognized dependency types

Projects using npm can use a structured format to list their dependencies, i.e., the npm packages they use. Four types of dependencies can be specified in package's metadata kept in a file named `package.json` in project's source directory. These types are described in Table 5.1. Throughout the rest of this work, the opposite of a dependency shall be called a **dependee**.

5.2 Statistical analysis of the npm ecosystem

To determine which projects using npm Registry are the most popular among developers, the dependency relations between packages were counted and analyzed. First, the metadata of all published packages in JSON format was downloaded. Download took

Table 5.1: Dependency types recognized by npm.

Metadata key	Meaning
`dependencies`	Packages needed at runtime.
`devDependencies`	Packages needed or useful for development, often minifiers/bundlers, test frameworks, linters, and version control integration tools.
`optionalDependencies`	Similar to `dependencies` but only needed for some additional functionality. npm supports installing a package without its optional dependencies, but by default it does install them.
`peerDependencies`	Used to specify compatible versions of tools for which the dependee is a plugin.

place on the days following April 14th, 2025. The metadata was processed to only include information about releases made after April 14th, 2025, yielding 3 519 767 package entries. Curl program was used to make requests to Registry's CouchDB view at https://replicate.npmjs.com/_all_docs. It is worth noting that this API endpoint's functionality has since changed and other means would be necessary to download the entire Registry metadata again in the future.

For the purpose of rankings discussed next, the dependees being multiple versions of a single package were counted as one. Similiarly, version constraints in dependency specifications were ignored.

5.2.1 The most popular dependencies – changes over five years

One of several metrics of package's popularity is its number of public dependees. Up to 2019 such a ranking of 1 000 packages most often specified as others' dependencies used to be published by Andrei Kashcha [Kas19]. A similar list computed from newer data for the purpose of this work was used to check how much the set of the most popular packages changed between August 2019 and April 2025. The goal was to find out how many of the previously popular projects keep to be chosen by developers and how many stopped being actively used, perhaps becoming legacy software. The overlap between the rankings is visualised in Figure 5.1. For each natural n in the range $[1, 1000]$, n most popular dependencies from both rankings were selected. The overlap of selected packages from first and second ranking was computed and plotted with the values of n on the X axis of the figure.

The "winners" in 2019 and 2025 were `lodash` and `react` with 69 147 and 263 038 dependees, repsectively. It can be seen that about 150 most depended packages form a

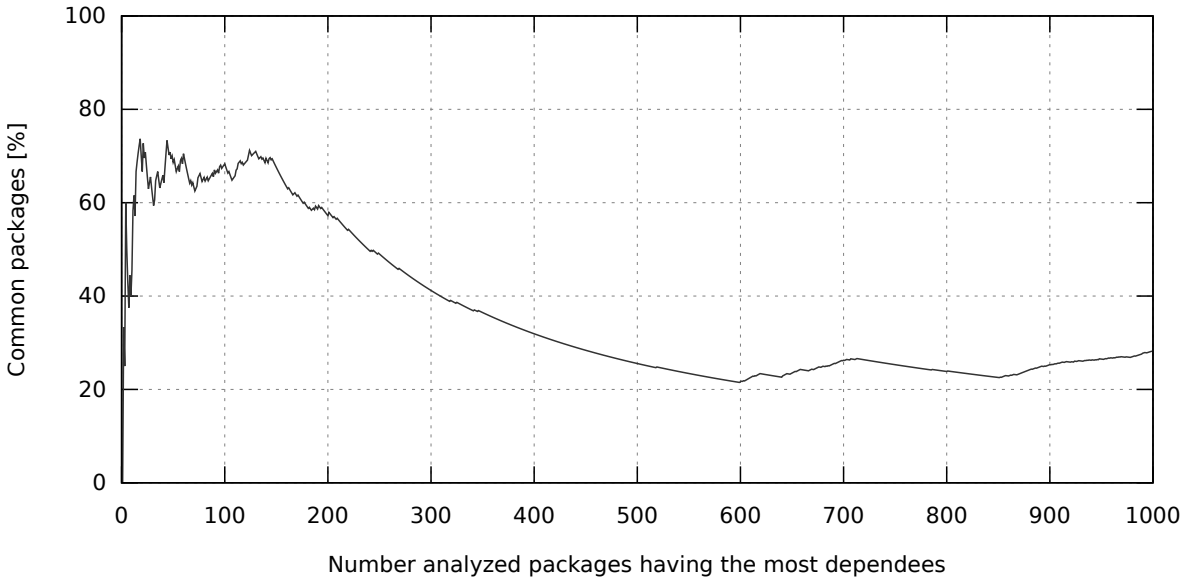

Figure 5.1: Overlap of the most popular npm dependencies from 2019 and 2025.

relatively stable forefront, with further part of the ranking having changed more over five years. Nevertheless, it is worth noting that certain packages with no new releases for several years still rank relatively high in 2025. Examples are `lodash`, `request`, and `q` ranking third, 17th, and 157th, respectively.

As a conclusion, if certain software is intended to be used for more than a few years, dependencies for it must be considered more carefully when they are not from among the 150 most popular ones. Otherwise, the risk of project's direct dependency becoming legacy software grows. However, often other characteristics of a package will determine whether it should be considered reliable. Ultimately, the matter of who maintains a package and how it could help the project are more relevant than a ranking position.

5.2.2 The most popular dependencies – popularity tresholds

The numbers of dependees corresponding to ranking positions can be used to infer some qualities of the ecosystem. This correspondence is presented in Figure 5.2.

In 2019, the 1000th most popular dependency package in the npm Registry had 346 dependees. By April 2025, the 1000th package in the ranking had already 4 771 dependees. This reflects the growth of the entire ecosystem, whose package repository had about one million packages in July 2019 and about 3.5 million packages in April 2025. However, this would by itself only explain a rise in dependee counts by about a ratio of 3.5. The

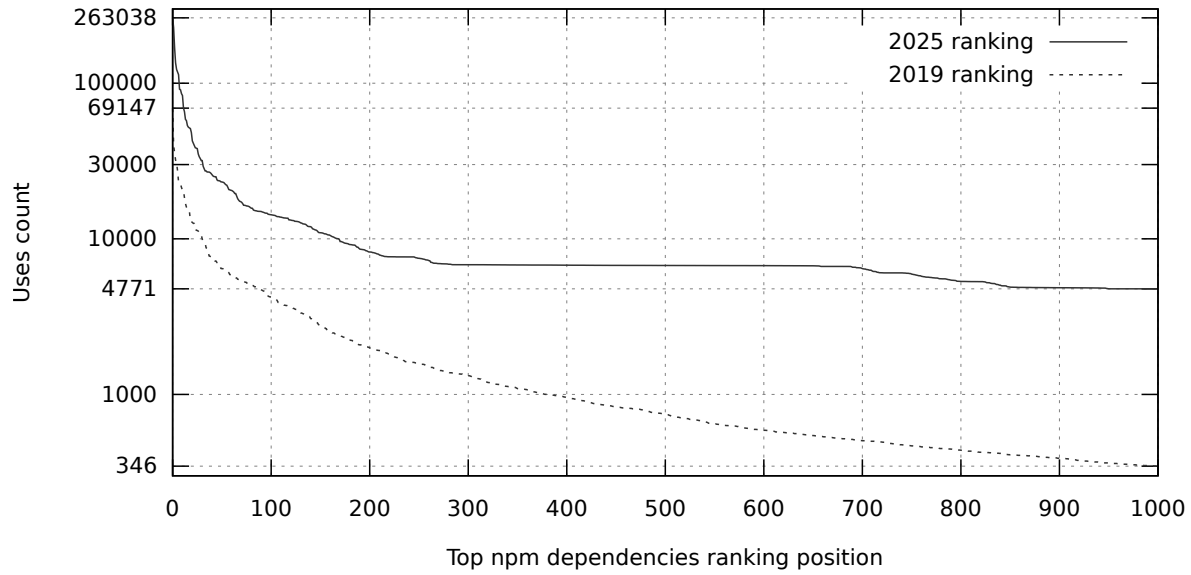

Figure 5.2: Number of packages using the most popular dependencies.

aforementioned increase from 346 to 4 771 dependees is over four times greater. This needs to be attributed to growing projects' complexity, as there is a tendency to use more dependencies. A plausible additional explanation is higher overlap of functionalities between packages, i.e., situations occur where multiple popular libraries exist for a single task.

5.3 The most popular development dependencies

In the context of supply chain security, the development dependencies are as important to research as the runtime dependencies. A popularity ranking similar to the previous one was compiled for packages occuring the most in the `devDependencies` collections of others. An analogous correspondence of ranking position to development dependee count is presented in Figure 5.3. No development dependencies ranking from 2019 was found that could be used for comparison. Instead, the runtime dependencies plot from Figure 5.2 was re-included for easier reference.

The first position belongs to `typescript` with 840 161 development dependees. A treshold of 2 185 development dependees needed to be reached by a package to be included in the ranking. The curve for development dependencies is steeper, meaning there is a clearer forefront. This in turn indicates that the functionality overlap mentioned in 5.2.2 is possibly a smaller problem in this case.

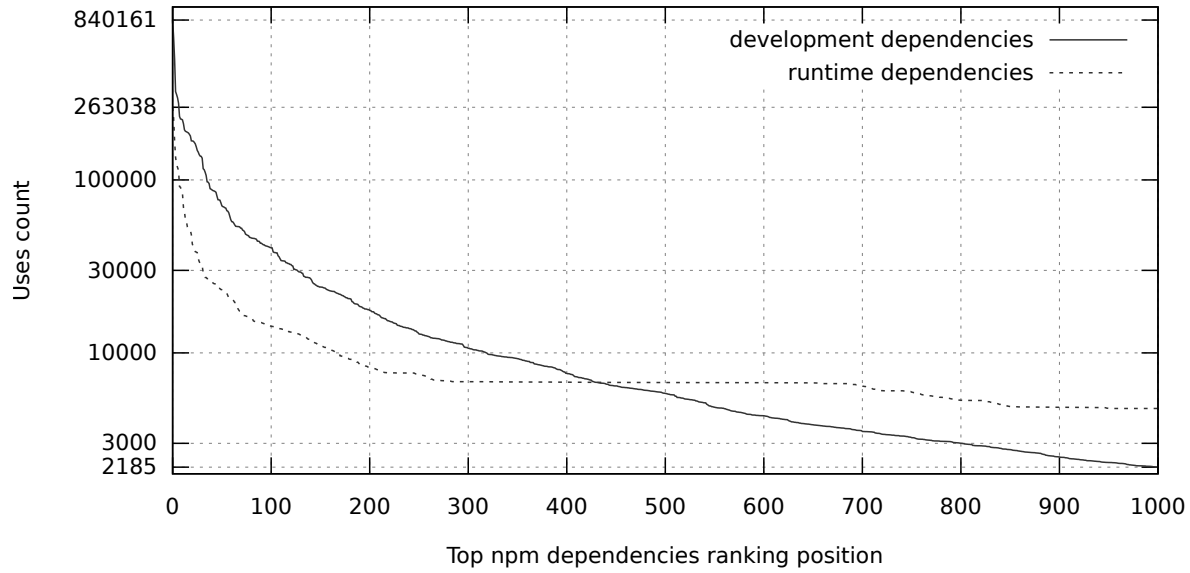

Figure 5.3: Number of packages using the most popular development dependencies.

5.4 Overlap of the most popular runtime and development dependencies

It is possible for a popular development dependency to also be specified as a runtime dependency by some packages. Realizing how often this happens can help judge whether certain kinds of issues are likely to occur in the ecosystem. The overlap of runtime and development dependencies is visualized in Figure 5.4, using the same approach as for the overlap in Figure 5.1 discussed in 5.2.1.

Since packages listed as `dependencies` are often libraries or frameworks and those listed as `devDependensies` are commonly applications, one could expect a smaller overlap than that of about 15-30% which was found. A possible explanation is that unlisting a package from `devDependencies` add instead including it among `dependencies` creates no major change for project developers. A command like `npm install` shall still resolve that dependency and include it in the environment it creates. It is therefore possible that a non-negligible number of dependencies is incorrectly categorized by the dependees.

It used to be a known fact that among packages listed as `devDependencies` there are many which are not needed to merely rebuild a project. These could be automatic code formatters, tools responsible for integration with version control, etc. and they could be eliminated from automated builds to make them lighter on resources and to decrease the

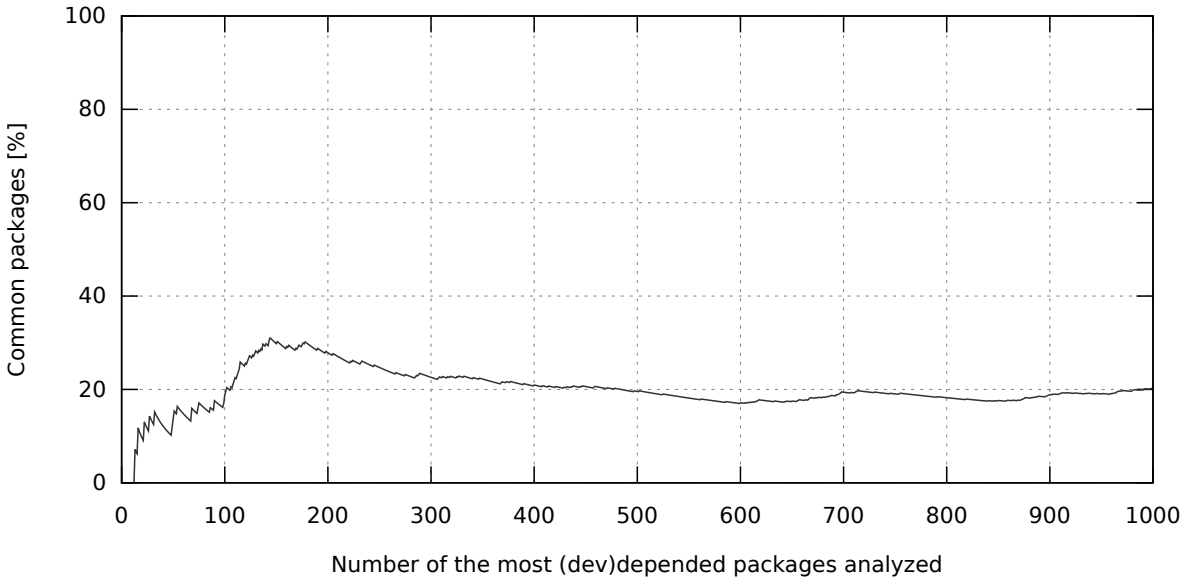

Figure 5.4: Overlap of the most popular npm runtime and development dependencies in 2025.

attack surface. Based on these results it is reasonable to expect that the similar holds for runtime dependencies. This provides a justification for experiments aimed at eliminating the extraneous dependencies without breaking the functionality of packages.

6. Possible paradigms for hermeticity and reproducibility

Certain popular software technologies – with npm being one of them – prove difficult to combine with existing reproducibility-focused workflows. Possible causes of this were discussed in 4. The package creation workflows of – largely reproducible – system software distributions Debian and GNU Guix were explained in 3.1.1 and 3.2.1, respectively. An explanation of the npm ecosystem followed in 5.

With all the above being considered, the ability to handle software with numerous dependencies – which can have complex relatonships – appears relevant to the goal of rebuilding parts of the npm ecosystem hermetically and reproducibly. Based on the knowledge gathered, possible approaches to hermeticity and reproducibility in the context of dependency resolution shall be critically analyzed. They shall be classified as distinct paradigms. The introduced paradigms shall be discussed in the context of security and their applicability to the build process of npm pacakges.

Paradigms 0 through 2 represent existing approaches. Paradigm 3 is an intermediate one that leads to 4, which is a generalization of the former. Paradigms 3 and 4 are an innovation originating from this work. They are meant to optimize the way build inputs are determined and also ensure that no unnecessary single points of failure are created which would not be secured through verified reproducibility. The new paradigms are suggested as bases for hypothetical new software packaging workflows that would make hermeticity and reproducibility easier to achieve with software from, among others, the npm ecosystem.

Kraków, 2025

6.1 Paradigm 0 – lack of actual reproducibility

If one is to build an npm package in the most basic way, with use of commands like `npm install` and without a pre-computed dependency tree, then network connectivity is necessary. Without it, the npm tool cannot download packages metadata from its repository, the npm Registry. But if network access is allowed, then the build – and therefore its result – might depend on downloaded code and data other than dependencies' metadata. Some commonly used npm packages require additional connections to function. For example, the `playwright-webkit` library, upon installation by npm, downloads executables from a third party server. That library is an intermediate test dependency of a popular web library, JQuery, used by about 74% of the most popular websites [W3JL].

Author of an npm package can assign so-called **distribution tags** to its specific versions. The tag can be though of as a string label that points to a package version. Tags can be used by dependees as an alternative way of specifying the depended version of the package. Some of the commonly used tag names are `next`, `beta`, and `latest`. When the developer publishes a new package version, the npm tool by default automatically assigns the `latest` tag to that version.

The build process of an npm project relies on downloaded metadata of packages. As a result, if a dependency author publishes its new version or alters the distribution tags, it might cause later rebuilds of the package to use a different set of inputs. The final result can be different, so it is not reproducible. Additionally, the repository constitutes a single point of failure because compromising the repository allows altering the served metadata of package versions. The compromised repository could, for example, spoof package's distribution tags or hide the existence of certain versions of a package, thus allowing only vulnerable versions to be used.

With files coming from a third party server, we have even less guarantee that they were not maliciously tampered with. A library that downloads such files during package build could verify them. For example, its authors could make the library contain cryptographic hashes of the required external files. The library could then check that every downloaded file has a matching hash. Unfortunately, we have no mechanisms to mandate that this kind of verification takes place in cases like that of `playwright-webkit`. This means ad-hoc file downloads are a bad security practice. They would need to be eliminated or restricted

for reproducibility to be leveraged.

Despite the above, reproducibility tests of npm packages are actually attempted, with one example being Pronnoy Goswami's research [Gos20]. It has to be noted that the results of such tests are likely to be unstable, yielding different results if repeated at a later date.

6.2 Paradigm 1 – inputs determined by human-maintained references

One of the possible methods of determining the set of build inputs is used by GNU Guix. Its package recipes contain references to dependency packages that have their versions predetermined. As a result, questions like "Should `foo` use its dependency `bar` in version 2.4.5 or 3.0.1?" are already answered. An update to a package definition results in the updated package being used everywhere that particular definition was referenced. The update takes form of a commit or commit series in the Git VCS and is subject to review by co-authors of the distribution.

If we fix a GNU Guix revision to be used, then in a package build – which is hermetic by design – all inputs are unambigiously determined. No repository of metadata can open the user to the risk of using incorrect dependency versions. In other words: the threat on the part of improperly defined dependency versions is of the same nature as that on the part of improperly written code. And – as users of any kind of software – we are deemed to accept threats of this nature.

Maintenance of this kind of system is, of course, more labor-intensive. Every alteration of a package recipe – also including software's version updates – is an update to GNU Guix' Git repository. Such update involves labor of distribution maintainers, similarly to Debian's case. A sample list of 26 consecutive commits to GNU Guix' repository – with 15 package updates among them – is presented in Listing 6.1. The changes in the list were made by multiple contributors during an hour between 12:00 and 13:00 on April 11, 2025. The changes are listed oldest to newets. Details of the newest one are additionally shown. A non-negligible amount of work is clearly needed to handle many changes manually. The great number of small changes might therefore lead to a yet unverified assumption that too big effort is required of distribution maintainers. If true, this could hamper the

growth of the software collection available through GNU Guix.

```
ff5181e27e * daemon: Do not make chroot root directory read-only.
661bfd5459 * gnu: Remove duplicated package show-me-the-key.
1300d15763 * gnu: emacs-fj: Update to 0.6.
80cda80489 * man-db: Parse man macro arguments better.
c705d6e035 * man-db: Support mdoc-formatted man pages.
d1a1d7f2f7 * gnu: Add julia-simpletropical.
772b70455d * gnu: cgit: Update to 1.2.3-9.994d3fe.
c2f2dd1bf8 * gnu: guix-build-coordinator: Update to 0-128.7a253d1.
2d17db72d8 * gnu: font-go: Update to 2.010.
6fc5fb97cb * gnu: font-adobe-source-sans-pro: Update to 3.052.
fff4c6462f * gnu: font-arapey: Add revision number.
7dc2151550 * gnu: font-carlito: Update to 0.0.0-1.3a810ca.
5cdfd3d81f * gnu: azimuth: Update to 1.0.3-0.050f838.
70aa3b9c2f * gnu: font-adobe-source-han-sans: Update to 2.004.
198fe8bcdf * gnu: emacs-vundo: Update to 2.4.0.
0c7ffaacd2 * gnu: emacs-parsebib: Update to 6.7.
a775db2460 * gnu: emacs-jinx: Update to 2.1.
33c3ee5985 * gnu: emacs-julia-mode: Update to 1.0.2-0.7fc071e.
172e9a1aa1 * gnu: emacs-magit: Simplify package.
d262248c55 * gnu: localed: Remove trailing #t and re-indent.
383f7f5c89 * gnu: localed: Modernize.
41c40bc1cf * gnu: localed: Update to 257.4.
098b5cdf9c * gnu: elogind: Update to 255.17.
c17c6b9820 * services/base: Remove extraneous UDEV_CONFIG_FILE environment variable.
dedeb90501 * gnu: eudev: Build with udevrulesdir pointing to /etc/udev/rules.d.
744e973de3 * gnu: samba/pinned: Update to 4.18.1.
1 file changed, 2 insertions(+), 2 deletions(-)
gnu/packages/samba.scm | 4 ++--

modified   gnu/packages/samba.scm
@@ -177,7 +177,7 @@ (define-public samba/pinned
   (hidden-package
    (package
      (name "samba")
-     (version "4.17.0")
+     (version "4.18.1")
      (source
       ;; For updaters: the current PGP fingerprint is
       ;; 81F5E2832BD2545A1897B713AA99442FB680B620.
@@ -186,7 +186,7 @@ (define-public samba/pinned
         (uri (string-append "https://download.samba.org/pub/samba/stable/"
                             "samba-" version ".tar.gz"))
         (sha256
-         (base32 "0fl2y5avmyxjadh6zz0fwz35akd6c4j9lldzp2kyvjrgm36qx1h4"))))
+         (base32 "03ncp49pfpzjla205y3xpb9iy61dz4pryyrvyz26422a4hpsmpnf"))))
      (build-system gnu-build-system)
      (arguments
       (list
```

Listing 6.1: List of consecutive changes committed to GNU Guix, with contents of the bottommost one included for reference.

In addition, many package managers following other paradigms can make use of permitted dependency version ranges declared by packages. This way npm, APT, and others can automatically avoid using incompatible dependency versions. However, Paradigm 1 does not allow such optimization to be employed.

It is worth highlighting that in GNU Guix the URLs and hashes that comprise identification data of program sources are maintained together with package recipes, as can be seen in Listing 6.1. As explained, this approach might have additional consequences in the amount of distribution maintainers' labor. Nonetheless, it can also positively or

negatively affect the chances of malicious sources being referenced in a recipe. This is an important supply chain issue to recognize, but it is independent from the concept of paradigms introduced in this chapter.

6.3 Paradigm 2 – reproducibility not applied to dependency resolution

The problem of the dependency resolution process being unreproducible was explained in 6.1. In this context, the actual package build can be partitioned into several distinct steps, for example

1. dependency resolution,
2. dependency installation,
3. code transformation/generation,
4. automated tests, and
5. installation/packing.

Steps 1 and 2 are sometimes performed together, for example as part of a single command invocation. However, in case of some package managers – including npm – the set of resolved dependencies with their versions can also be recorded for later reuse. It is done with so-called **lockfile** – a file that project developers can add to a VCS and which allows dependency installation to be repeated without re-downloading metadata nor re-running the resolution algorithm. In npm projects this file is saved as `package-lock.json` or `npm-shrinkwrap.json`.

With a precomputed `package-lock.json` we can therefore download the dependencies and use them as inputs of the hermetized build, narrowed to steps 2-5. Upstream software's original build procedures sporadically expect network access during these steps. The build process of the aforementioned JQuery is one of those few cases where this occurs. Such problems would need to be corrected manually, in package recipes of a hypothetical distribution applying Paradigm 2 to a broader population of npm packages. A typical solution in, e.g., Debian is a patch that eliminates such access attempt or replaces it with a reference to a local, formally-approved input.

If npm project authors fail to provide an appropriate lockfile – which can happen – it could be generated by one of the parties that rebuild the software. Step 1 would then need to be performed unhermetically, with network access. The obtained `package-lock.json` would then be treated as additional build metadata, distributed to the other parties. When a build were to be repeated to verify the reproducibiliy of the result or for other purposes, presence of this metadata would be required.

The benefit of Paradigm 2 is that one can proceed in achieving reproducibility of most of the build process and further leverage it. In fact, comments in the source code of JQuery indicate that its developers – to some extent and with disregard for possible changes in files being downloaded during the build process – did actually work on making JQuery's build process deterministic when the `package-lock.json` is used.

The main disadvantage of Paradigm 2 is that dependency resolution is still not secured by hermeticity nor reproducibility. Even when changes to project's `package-lock.json` take the form of version control system commits, these are unlikely to be reviewed as carefully as ordinary software code changes. Dependency trees can be complex. `package-lock.json` files counting over 1000 entries are common. As a result, the shape of a particular resolved dependency tree is difficult to explain without additional tools.

The described approach requires generalization to building a project that uses multiple repositories, e.g., npm Registry + Python Package Index + Rust Package Registry. That is because multiple dependency trees from multiple software ecosystems are involved. Theoretically, even in terms of a single ecosystem and a single repository, we might need to resolve multiple sets of dependencies in step 1. In effect, an actual collection of lockfiles would need to be treated as the aforementioned additional build metadata.

6.3.1 Debian implementation

Interestingly, a variant of Paradigm 2 can be found in Debian, which is considered one of the most reproducible software distributions. That is because the package recipe shared as `debian.tar.xz` file contains the names of direct build dependencies but not necessarily their precise versions nor the indirect dependency names. It is actually the `.buildinfo` files where the published packages' build environments metadata can be found. Much of this metadata is determined by the dependency resolution process, as performed by Debian's APT tool during the initial Debian package build.

Although this does formally fall into the scope of Paradigm 2, Debian packagers' perspective is still similar to that of Paradigm 1 users. That is because – as explained in 4.4 – a single Debian release typically only advertises a single version of a given package at any point in time. Unless multiple Debian releases are mixed together, this makes the input metadata of APT's dependency resolution process flat. This, in turn, makes packagers ultimately responsible for ensuring version compatibility between packages in this flat space.

6.4 Paradigm 3 – deterministic dependency resolution inputs ensured

For our dependency trees from Paradigm 2's step 1 to be secured through reproducibility, we need to be able to repeat the dependency resolution step using the same data about candidate dependency packages. Neither `.buildinfo` nor `package-lock.json` files preserve all metadata actually consulted by the resolution algorithm. They lack information about packages that were considered but rejected as final dependency tree members. As such, full dependency resolution cannot be performed based on just these files' contents. It can be argued that the risks this causes for Debian are small because the general public cannot create new packages that could then be immediately used as dependencies. Here, one of the most likely dependency resolution attack scenarios involves supplying the build with an outdated, faulty compiler package already present in the distribution. One theoretical attack utilizing a compiler bug was described in [BCR15]. In contrast, manipulation of `package-lock.json` in an npm package build can more easily lead to an attacker-published package being installed in the build environment.

In case of npm projects, one of the simplest solutions would be pointing the npm tool to a local mock of a repository server, speaking the HTTP protocol. The mock would function as a proxy that downloads required packages' metadata from the original npm Registry server, alters it and returns it as responses to the npm tool's requests. Each response – containing the metadata of all versions of a single npm package – would be filtered not to include the versions of packages that were published after a chosen time treshold. The treshold could be, e.g., the release date of the project version being built. In repeated build attempts, the relevant metadata served by mocked registry ought not

to change. Corner cases shall occur, in form of dependencies being removed from the official registry due to copyright claims or in form of projects' dependence on particular developer-alterable distribution tags of npm packages. These problems should be rare enough to be fixable manually or with reasonable defaults. For example, a mock `latest` tag could be attached to the newest version of each npm package whose metadata is served.

This approach does not completely eliminate the threat of the dependency resolution process being maliciously influenced. In particular, the packages' metadata could be maliciously modified even earlier, for example as a result of the official registry's infrastructure being compromised. However, compared to Paradigm 2, the number of moments when malicious modifications could occur is decreased. Similarly, the scope of what could be modified is more limited. To decrease the changes of the hypothetized attack on the registry being successful, additional means of detection and mitigation could be employed. For example, trusted third parties can serve as "canaries", publishing cryptographically signed information about what package metadata was being served by the repository as of given date. The initial builder can also record the resolution metadata and make it available to rebuilders, effectively acting as one of the suggested canaries. The holy grail of avoiding a single point of failure – one in the form of a centralized registry – would be deriving the resolution metadata of packages from those packages themselves once they are also rebuilt locally. This would present a bootstrapping challenge that – when solved – would open the way to dependency resolution without major reliance on any centralized service.

Regardless of the employed approach to securing the dependency resolution inputs, the actual concept of Paradigm 3 is to make the inputs plausibly deterministic and then repeat the dependency resolution process upon every repetition of a given package build. The remaining steps of the build process are performed analogously to those in Paradigm 2. The issue of generalization to projects utilizing multiple repositories is also analogous to that in Paradigm 2.

6.5 Paradigm 4 – hermeticity relaxed and deterministic dynamic inputs allowed

One can notice that in paradigms 2 and 3 the first step, dependency resolution, is treated different from the subsequent ones. The result of step 1 is a collection of one or more lockfiles that identify dependencies' files, e.g., through names and versions or URLs and hashes. A tool that implements a given paradigm would need to – between steps 1 and 2 – prepare an appropriate isolated environment for package build, for example a Linux container. Lockfile-identified dependencies would need to be exposed inside.

In Paradigm 3, the initial download of packages metadata can happen through a locally-run mock of a repository server. I.e., the isolated dependency resolution process has a service perform possibly hermeticity-violating actions on its behalf. Yet, care is taken to make the results of those actions deterministic. Paradigm 4 extends this approach to all steps of the build. The installation step, knowing project's recursive dependencies identified by the lockfiles from step 1, could have the service supply the dependencies' files into the otherwise isolated build environment. There is no more need to provide separate isolated environments to two different parts of the build process – step 1 and the chain of remaining steps. As long as the hermeticity-violating actions performed by the service on build's behalf are deterministic, this should not make build results less reproducible. The process can be thought of as **eventually-hermetic**, bacause repeated builds are likely to request the exact same actions, requiring the same external data, which could be cached and reused, making subsequent runs network-independent. At the same time, this approach **removes the need of having all build inputs identified in advance**, simplifying the entire build process.

Let us provide another example of an action that could be deterministically carried out on behalf of the build – checking out of a Git repository revision. Under Paradigm 4 this could happen through the hermeticity-violating service, making the repository a **build input determined dynamically**. If the checkout operation uses a repository URL and, e.g., a Git tag[1], it is by itself not deterministic – result can vary in time, for example due to tags being changed in the upstream repository. In this case, additional means – like those already mentioned – would be needed to ensure the determinism of the

[1] unrelated to npm package's distribution tag and not to be confused with it

checkout action. However, no such extra measures are necessary if the checkout operation uses a commit hash made with an algorithm deemed cryptographically secure. Preimage attack resistance is of practical relevance, making even SHA-1 applicable as of 2025.

This approach makes the logic of an eventually-hermetic package build more straighforward. If, for example, step 3 required an extra resource or tool, in paradigms 1-3 that requisite would need to be identified beforehand. Under Paradigm 4 this is not necessary.

6.5.1 Security-oriented justification

How secure would the Paradigm 4 be? Its security relies on the viability of employed means of ensuring the determinism of dynamic inputs. A GNU Guix-like approach of maintaining the cryptographic hashes of all downloadable resources in a VCS is possible. While the collection of resources still needs to be identified in advance of any build, there is no more need to record exactly which ones are needed for which particular package build – by itself a huge simplification. This makes Paradigm 4 no worse than – seemingly the most secure – Paradigm 1, as implemented in GNU Guix.

However, the concept of paradigms – as introduced by this work – is not strictly dependent on the way of ensuring the integrity and determinism of software sources and of other build inputs. The approach of keeping hashes of packages' sources embedded in code kept in a VCS can be criticized. In theory, changes to the version-controlled recipes code – with input resources' hashes – are subject to review. However, despite the positive security aspects of human-conducted code reviews, such system makes it easy for reviewers to lose vigilance – especially when facing a "flood" of package recipe updates, as shown in Listing 6.1. Some could argue that it would be beneficial to completely replace the version-controlled hashes of sources with a network of canaries that record the tagged revisions found in VCS repositories and the contents of software's published release files. This aproach is applicable to Paradigms 4, 3, and 1 alike. It simply happens not to be employed by GNU Guix as of 2025.

7. Automated package builds experiment

The previous chapters of this work has lead to the following hypotheses and questions.

HYPOTHESIS 1. The dependency tree sizes of npm packages and acceptance of conflicting dependencies by the platform appear to be the major sources of difficulty in packaging the npm ecosystem in reproducibility-focused distributions. Is this truly the main factor?

HYPOTHESIS 2. Re-generation of npm lockfiles – an operation necessary for the security improvement offered by proposed paradigms 3 and 4 over Paradigm 2 – is expected to rarely cause npm package build failures which would not occur with developer-supplied lockfiles. Are such failures indeed uncommon?

HYPOTHESIS 3. As speculated in 5.4, both direct and indirect dependencies of npm projects are often unnecessary. Is this indeed the case?

QUESTION 4. What are the typical sizes of dependency trees needed to build npm projects and how much can these trees be typically shrinked?

QUESTION 5. How often do dependency conflicts actually occur in npm dependency trees and how often – or to what extent – can they usually be forcibly eliminated without causing breakage?

QUESTION 6. Are forced removals of npm project's dependencies and forced elimination of dependency conflicts likely to cause non-obvious breakages that only become apparent when seemingly successfully-built package turns out to be disfunctional or nonfunctional? If so, how best to avoid them?

HYPOTHESIS 7. npm projects' dependencies are seldom specified by distribution tags and removal of distribution tags from npm dependency resolution metadata, with

automatic addition of a mock `latest` tag as mentioned in 6.4, is expected to cause few dependency resolution failures. Is this a valid assumption?

QUESTION 8. Can we deliver a prototype that performs npm project's dependency resolution as proposed in paradigms 3 and 4?

To verify and answer these, an experiment was conducted which involved automated build attempts of top npm projects, selected by their position in the npm Registry package rankings. The selected set consisted of projects belonging to the first 200 of either the most popular `dependencies` or `devDependencies` as of April 14th, 2025. Due to some overlap between the rankings, the actual size of the set was 329. The build procedure, described in the following subsection, was designed with the help of a trial-and-error approach.

7.1 Method and environment

The experiment was conducted on an x86_64 machine. For details, see Listing 7.1. All operations were performed under version 5.15.0 of the Linux kernel. All filesystem operations were backed by an ext4 filesystem. No effort was made to employ means like disorderfs, described in [LZ21], because this work is not concerned with eliminating the traditional sources of nondeterminism.

```
Packages:
        0: Intel Core i7-7500U
Microarchitectures:
        2x Sky Lake
Cores:
        0: 2 processors (0-1), Intel Sky Lake
        1: 2 processors (2-3), Intel Sky Lake
Clusters:
        0: 4 processors (0-3),  0: 2 cores (0-1), Intel Sky Lake
Logical processors (System ID):
        0 (0): APIC ID 0x00000000
        1 (1): APIC ID 0x00000001
        2 (2): APIC ID 0x00000002
        3 (3): APIC ID 0x00000003
```

Listing 7.1: Details of the processor used during the experiment, as reported by cpuinfo utility.

The diagram in Figure 7.1 describes the flow of activities during testing of a single npm project. In the diagram, start and end are denoted by a filled circle and a circle with a white ring inside, respectively. Flow branches are represented by hexagons and merges – by rhombuses. The particular operations present in the diagram are described further below.

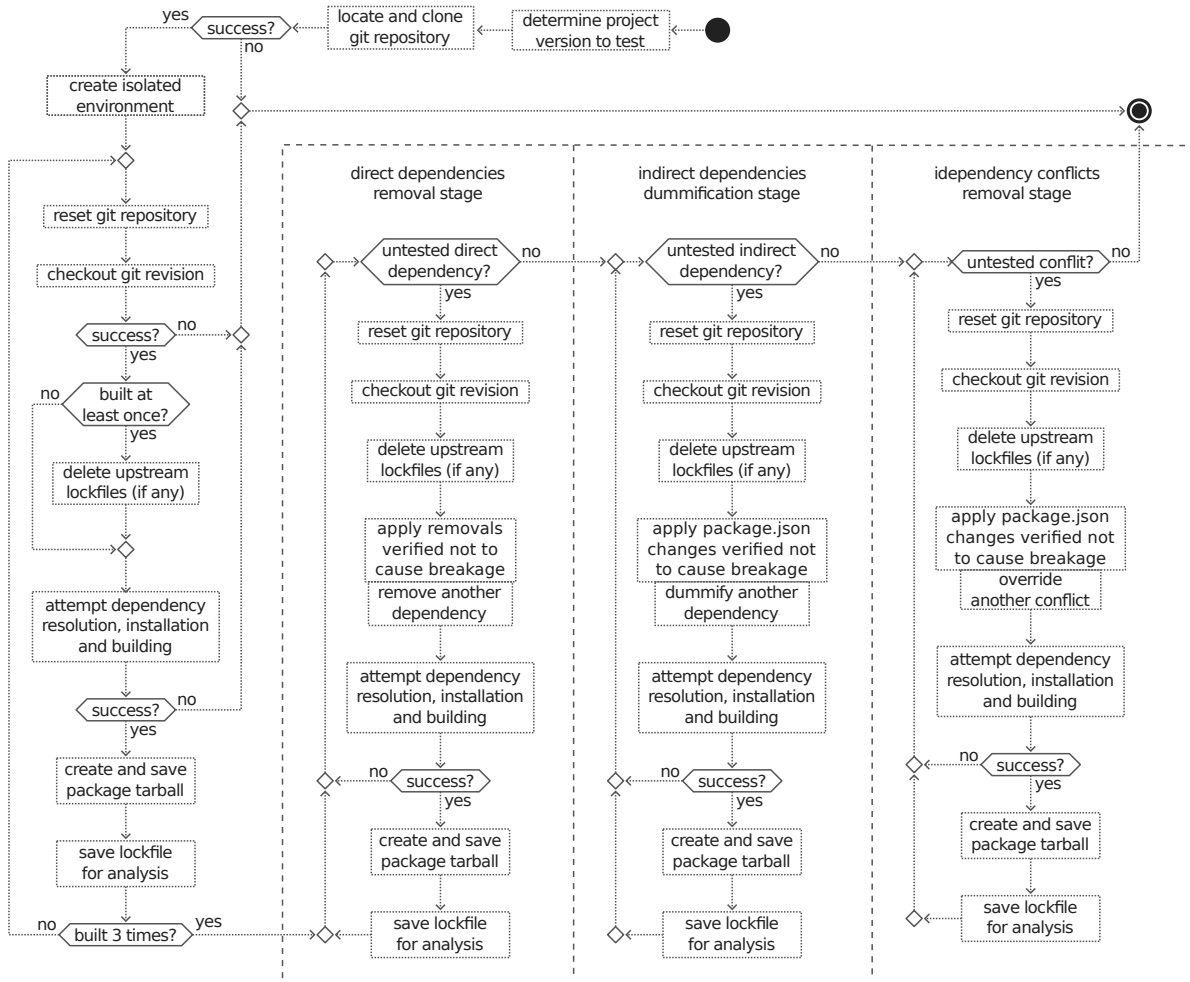

Figure 7.1: Activity diagram describing the experiment as performed on each tested npm project.

7.1.1 Containerized environment creation

For each tested project, the version to build was first selected as its highest non-pre-release package version published before April 14th, 2025. Versions' publishing dates that were consulted were part of packages' metadata downloaded from the npm Registry. For the selected version, the relevant Git repository URL and – where possible – release's Git commit hash were extracted from the available metadata. The URL was sometimes present with a `git+` or `git:` prefix, which had to be dropped. The repository at learned URL was then cloned to a local directory, with submodules included through the use of Git's `--recurse-submodules` option.

Upon successful retrieval of source repository contents, a semi-isolated environment, based on Linux containers, was created. Upon creation, the environment only had access to a minimal collection of software. It comprised

- Node.js runtime including bundled npm application,
- Git version control tool,
- GNU Bash shell, which served as the POSIX-compliant shell used by `exec` function of Node.js,
- GNU Coreutils,
- GNU which, invoked by experiment's code,
- GNU Guile and several Guile libraries, being the driver for the experiment's code, and
- dependencies of the above, e.g., a C library.

The environment was created as a container shell managed by GNU Guix. The version of GNU Guix used was built from Git revision `143faecec3`, the last one before April 14th, 2025. It featured Node.js runtime in version 22.14.0, npm in version 10.9.2, and Git in version 2.49.0. Inside the environment, the applications listed above were available through the `PATH` variable and also symlinked under `/bin` directory through the use of `--emulate-fhs` option of `guix shell`. The environment had no direct access to outside network, allowing us to state that experiment's results reflect the behavior of a hermetic build process. Network isolation also helped make sure that no dependency was installed "on the side", without being recorded in project's lockfile. The environment was also isolated filsystem-wise, with specially prepared directories shared between the environment and the host. They were shared read-write or read-only, according to needs. Shared directories allowed container's guest to

1. access the code used to drive the experiment,
2. access npm project's cloned repository,
3. request npm packages' metadata and files from the host through named fifos,
4. receive them as files, and
5. persist lockfiles and final package files generated during build, for later inspection.

7.1.2 Source code checkout preparation

Inside the just-created environment, a checkout to npm project's appropriate revision was attempted in the following ways.

1. By switching to the commit hash previously extracted from the package metadata.
2. By switching to a Git tag identical – as a string – to the version being built.
3. By switching to a Git tag identical to the version being built, prefixed with letter "v". E.g., for a project version 3.2.2 a switch to Git tag `v3.2.2` would be attempted.

This sequence of tries was chosen based on findings from initial manual experiments and also on prior knowledge of common developer practices. In particular, it was found that a Git commit hash is not always advertised for a given npm package version. When it is, it sometimes corresponds to a revision that was never pushed to project's public repository. This appears to be most often caused by an automated release publishing software that makes a local commit as part of its operation. It was decided that in both cases – of the unknown and nonexistent commit hash – it is best to fall back to probable Git tags.

If directories named `node_modules` or `dist` existed in a successfully checked-out source repository, they were deleted before the actual build attempt. These directories are used to store npm project's installed dependencies and generated files, respectively. Although they are sometimes checked into version control, they are not sources per se and a hygienic npm project build should be performed without them.

It is worth noting that every build was conducted inside a full git repository checkout, with access to the `.git` directory containing project's history. This is unlike the practice of GNU Guix, Debian, and many other distributions where build inputs typically do not include any version control metadata. The decision was made based on the following considerations.

1. Build procedures most often rely on version control metadata for side tasks like generation of software authors list. These tasks are not highly relevant to our stated questions, but their failures could decrease the number of successful package builds that we seek to further analyze.

2. In actual distribution software packaging, the build process' reliance on version control metadata is considered easy to solve compared to the issues of dependencies.

3. While availability of version control metadata could theoretically ease smuggling of backdoor code in XZ-style attacks, it would be hardly practical – the backdoor would need to be somehow retrieved from version control history and invoked, in a hard-to-notice way. Building with full version control metadata is therefore a secure enough approach to be suggested for adoption by distributions.

7.1.3 Dependency resolution in a network-isolated environment

Dependency resolution was performed with the help of a dummy `npm uninstall` command, as shown in Listing 7.2. The options used made npm

- refrain from attempting network requests unrelated to the actual dependency resolution,
- refrain from actually installing the resolved dependencies or running their hooks,
- update project's lockfile to the current format if an older one was encountered, and
- make requests to a local mock of npm's repository.

The command either created the `package-lock.json` file from scratch, wrote a new version of it based on an existing lockfile found or left it unchanged. The latter happened whenever the existing `package-lock.json` was already in sync with dependency constraints specified in project's `package.json`. It can be noted that npm would also automatically use an `npm-shrinkwrap.json` file over `package-lock.json` if the former were present. However, this was not the case for any of the npm projects tested.

```
npm --no-progress --no-update-notifier \
    --audit false uninstall \
    --ignore-scripts --package-lock-only \
    --package-lock-version=3 \
    --registry=http://localhost:8080 \
    Experiment-Dummy-Package-To-Delete
```

Listing 7.2: The npm command used to produce an up-to-date lockfile.

For projects that utilize workspaces, attempt was made to also add workspace-related options `--workspaces`, `--include-workspace-root`, and `--workspace` as appropriate to `npm uninstall` and

subsequent npm invocations. Workspaces are a feature that allows multiple subprojects to be developed in subdirectories of a single npm parent project, with the parent and each subproject having its own `package.json` file. Despite the effort, all workspaced projects that were tested failed to build for other reasons. Several tested projects were found to use workspaces for smaller satellite utilities while having the project's package described by the `package.json` file in the root directory of the repository. Those were built without any workspace-specific npm options.

A minimal server, written for this experiment, listened for HTTP requests on port 8080 inside the network-isolated build environment. It received npm package metadata requests and passed the requested package names via a fifo to a service running outside the container. The service downloaded the metadata and filtered it to only contain information about package versions published before April 14th, 2025. The original npm distribution tags were stripped, but a mock `latest` tag was added for every package, pointing at its newest non-pre-release version from before April 14th, 2025. Pruned pieces of metadata were supplied to the guest-side server for use as responses to the npm tool.

7.1.4 Remaining build steps

After successful dependency resolution, packages listed in the lockfile were installed with the help of an `npm ci` command, as shown in Listing 7.3. Analogously to the dependency resolution step, package file requests were sent through the HTTP protocol to the local port 8080 and were handled by the guest-side server, which in turn relied on the host-side service to perform the actual downloads on guest's behalf.

```
1 npm --no-progress --no-update-notifier \
2     --audit false ci --registry=http://localhost:8080
```

Listing 7.3: The npm command used to install dependencies.

Successful installation of dependencies was followed by an invocation of a `build` action that npm projects conventionally define in their `package.json` files. The command used is shown in Listing 7.4.

```
1 npm --no-progress --no-update-notifier \
2     --audit false run build
```

Listing 7.4: The npm command used to invoke project-specific build operations.

At this point, the `package-locks.json` file was copied to a subdirectory of the results directory to be persisted after experiment's end. The same was done at subsequent builds, described further below, which involved modifications to the dependency tree. The collected lockfiles later allowed calculation of dependency tree sizes. When dependency tree modifications were found to cause changes to the built package, these lockfiles were also useful in understanding the exact reasons behind those changes.

As of April 14th, 2025, built npm packages are distributed as `.tgz` archive files. In the jargon they are called "tarballs" and in case of npm packages they are compressed using gzip algorithm. An `npm pack` command exists which can produce such archive from project's files. Although the same could be achieved with a traditional tar program, the npm's command is convenient, because – among others – it automatically omits unneeded files like the `node_modules` directory. The exact form of the command used to persist the built package is shown in Listing 7.5.

```
1 npm --no-progress --no-update-notifier \
2     --audit false pack \
3     --pack-destination=/PATH/TO/RESULTS/SUBDIR
```

Listing 7.5: The npm command used to create the built package file.

7.1.5 Repeated builds without upstream lockfiles

For a project that was successfully built with the described procedure, the process was repeated with alterations. Upon each repetition, the repository was brought to a clean state and had added Git hooks – if any – removed. However, re-creation of the entire semi-isolated environment was deemed unnecessary for the purpose of the experiment. After the repository was cleaned, each repeated build started with the Git revision checkout attempts described earlier.

The first alteration of the build was the removal of existing lockfiles recognized by npm or its alternatives: `npm-shrinkwrap.json`, `package-lock.json`, `yarn.lock`, and `pnpm-lock.yaml`. It happened right after the removal of version-controlled `node_modules` and `dist` directories. The removal of lockfiles was done to force a full dependency resolution. If successful, the build in this form was performed twice to check if dependency resolution and lockfile generation in npm suffer from obvious nondeterminism issues.

Additionally, all later builds of the project also involved the removal of existing lockfiles, besides other alterations.

7.1.6 Elimination of unnecessary direct dependencies

Each project known to build successfully with and without the removal of its version-controlled lockfile – if any – was tested further. The experiment checked whether it had the ability to build with each of its direct dependencies – tried in reverse alphabetical order – removed. E.g., a project with nine direct dependencies specified in its `package.json` – including those listed as `dependencies`, `devDependencies`, and `optionalDependencies` but not `peerDependencies` – was built nine times, each time with another direct dependency removed for the first time. The build was considered successful when the npm commands all finished with zero status. For each such successful build the tested dependency was recorded as unnecessary and was also removed in all subsequent build attempts, together with the dependency tested in a given attempt. E.g., if five out of first eight dependencies were found to be unnecessary, then subsequent build was performed with the ninth dependency plus the initial five removed. I.e., a total of six dependencies were removed in that build.

The removal consisted of erasing of dependency's entry in project's `package.json` file right after lockfiles deletion. However, the original `package.json` contents were always recorded and restored before the `npm pack` invocation. This was done to have the built package tarballs – each of which contains a copy of the `package.json` – easier to compare for other differences. Interestingly, for some projects the `npm pack` did not place the `package.json` inside the tarball verbatim and instead generated a variant of that file with some fields changed in a way custom to the project. One such case, concerning the `@testing-library/user-event` package, is discussed in 7.5.4.

All later builds of the project also involved the removal of dependencies identified at this point and the described restoration of the original `package.json` file.

7.1.7 Elimination of unnecessary indirect dependencies

With all apparently-unnecessary direct dependencies identified, the remaining indirect dependencies were tested. For it is unstraightforward to forcibly remove an indirect dependency from npm project's dependency tree, a choice was made to instead attempt

"dummifying" it. The npm feature of overrides – mentioned in 4.4.2 – was used to force the npm's resolution algorithm to always select a mocked, dummy version "0.0.0-msc-experiment-dummy" of a given dependency. At the same time, for the dependency package meant to be dummified, the local server providing packages' files and metadata on port 8080 would not respond with that package's real metadata. Instead, it would give a response indicating that the only available version of that package is the dummy version, which has no own dependencies. Additionally, it would serve the corresponding dummy package tarball with only minimal contents.

This process of identifying project's unnecessary indirect dependencies was analogous to that concerning direct dependencies. It involved multiple builds – more than a thousand in case of one npm project tested. In each build a single tested indirect dependency – together with the unnecessary indirect dependencies identified previously – was dummified. Each time the overrides were added to project's clean `package.json` file. The addition of overrides was carried out together with the removal of unnecessary direct dependencies from `package.json`. Build's all npm commands had to finish with zero status for the tested dependency to be assumed dummifiable. Each time the `package-lock.json` from the last successful build was consulted to determine the next dependency to test. Applicable dependencies were tried in reverse alphabetical order.

All later builds of the project also involved the dummification of indirect dependencies identified at this point. During the entire experiment, whenever a dependency to override already had an upstream override specified in `package.json`, the original override was being removed.

7.1.8 Elimination of dependency conflicts

Even after the elimination of unnecessary direct and indirect dependencies, project's dependency tree could still contain extraneous conflicting dependencies. Subsequent builds were carried out to forcibly remove those conflicts where possible, utilizing overrides. For every dependency that occured multiple times in multiple versions in the tree, a build attempt was made with an override which forced that package to be always used in the same, single version.

- If it happened to be both a direct and indirect dependency of the project – it was overriden with the version that was previously used to satisfy project's direct

dependency.

- If it was only an indirect dependency – it was overriden with the highest of the versions in which it previously appeared.

Just like before, the build was repeated to identify every dependency conflict that – when forcibly removed – does not cause any npm invocation finish with non-zero status.

7.2 Build attempt results

Two projects were found not to actually exist as real pieces of software. I.e., their npm packages were placeholders. Another 18 projects could not be tested due to limitations of experment's environment – they used dependency packages that are distributed through servers other than the official npm Registry. This made the npm tool attempt downloading these directly, which failed in a network-isolated environment. The results from build attempts of the final 309 projects are presented in Figure 7.2. Different types of failures were classified based on the first error reported in the build attempt. It means that, e.g., a project with unresolvable dependencies and a missing `build` action was classified as failing at the dependency resolution step.

7.2.1 Projects whose source repositories failed to be cloned

For projects in this category, sources could not be automatically retrieved. Either no repository URL was included in published npm metadata of the package version or the published URL was not valid.

Some packages were found to provide SSH URLs to their projects' GitHub repositories. Such URLs could not be used for anonymous cloning, despite the repositories themselves being – at least in some cases – public and anonymously cloneable through HTTP. A sample Git error message is presented in Listing 7.6. It was printed upon an attempt to use the `ssh://git@github.com/sinonjs/sinon.git` URL in a `git clone` command.

There was also a single case where an HTTP URL pointed at a repository that no longer existed. Other interesting unworkable URLs were ones with a branch name appended[1]. Some unworkable URLs were also pointing to web pages of repositories' sub-

[1] e.g., `https://github.com/emotion-js/emotion.git#main`

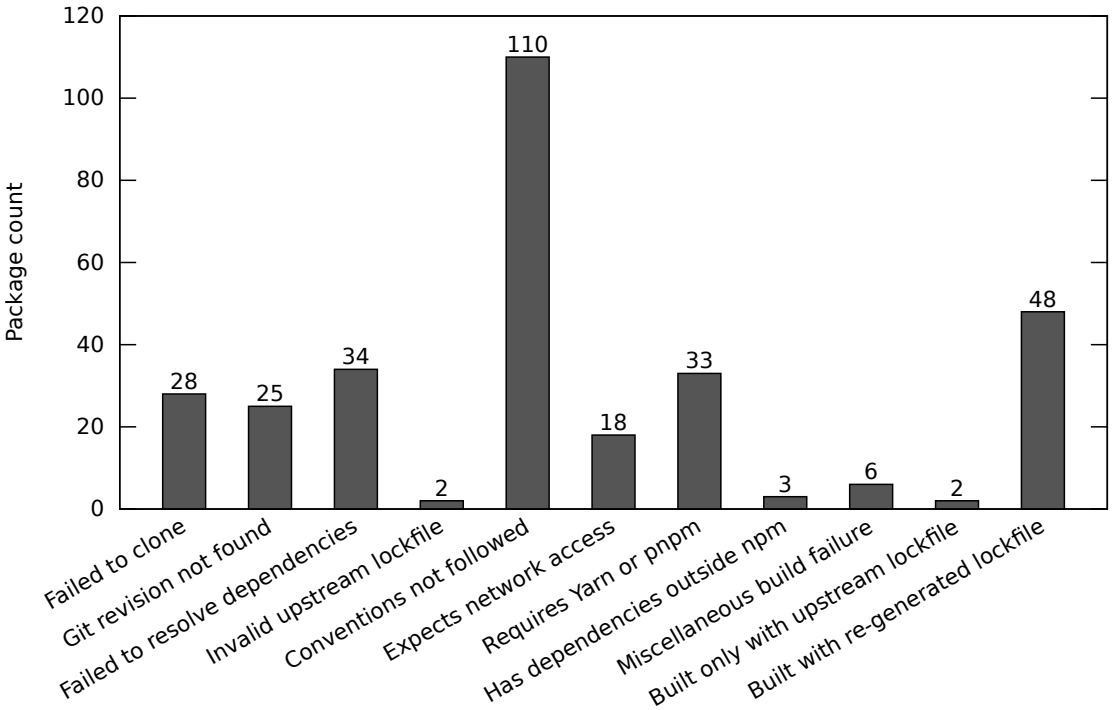

Figure 7.2: Statuses of automated hermetized build of top npm projects.

```
Cloning into 'new-checkout'...
Host key verification failed.
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

Listing 7.6: Error reported by Git upon an attempt to clone a repository using an SSH URL.

directories[2]. In a vast majority of cases a correction of URL with the help of a simple regular expression could be attempted. Interestingly, none of the tested projects were found to use a VSC other than Git.

7.2.2 Projects whose relevant source control revisions were not found

For projects in this category, the git source repository could be cloned but it contained neither the commit specified in package's metadata nor a tag corresponding to the version number being built. Reasons included

- VCS revisions being tagged differently, e.g., `PACKAGE-NAME@VERSION`,

[2] e.g., `https://github.com/babel/babel/tree/master/packages/babel-core/`

- particular version's tag being missing, and
- tags not being used altogether.

7.2.3 Projects that do not follow the conventions

Projects in this category either lacked a `package.json` file in repository's root or lacked a `build` action.

Size of this category seems to suggest that the conventions which we and others [Gos20] rely upon are very loosely followed. However, some projects classified here are trivial ones that simply do not require any operations to be performed as part of a `build` action. For example, package `semver`, as distributed through the npm Registry, was found to only contain files that are present in its project's source repository. I.e., none of the files were created or modified as part of the build process performed by that project's developers. The files in `semver`'s built package archive were found identical to those in the relevant revision of the source repository, with the repository additionally holding some other files, e.g., test scripts. `semver` does indeed not require compilation nor similar build steps. It has no need for a `build` action and therefore does not have one specified in its `package.json` file.

7.2.4 Projects with dependency resolution failures

Projects in this category had the `npm uninstall` command fail to create or update the lockfile. The predominant source of failure is related to peer dependency resolution, with a sample error message shown in Listing 7.7. Simplifying, peer dependencies are a feature through which developers can forbid npm from creating a dependency conflict with a particular package. Typically, an add-on package specifies its base package – which it enhances – as its peer dependency. If the base package were specified as add-on's casual dependency, npm's resolution algorithm could make the add-on package use its own copy of that base package. This is typically not the behavior the developer or user wants. Peer dependencies are a mean to prevent it.

The exact behavior of peer dependencies changed through the history of npm. One alternative package manager for the npm ecosystem – Yarn – is also known for behaving different than npm in some situations. It is suspected that most of the projects in this

```
npm error code ERESOLVE
npm error ERESOLVE could not resolve
npm error
npm error While resolving: typedoc@0.22.10
npm error Found: typescript@4.7.4
npm error node_modules/typescript
npm error   peer typescript@">=2.7" from ts-node@10.5.0
npm error   node_modules/ts-node
npm error     ts-node@"^10.2.1" from typescript-json-schema@0.53.0
npm error     node_modules/typescript-json-schema
npm error       dev typescript-json-schema@"^0.53.0" from the root project
npm error   dev typescript@"4.7.4" from the root project
npm error
npm error Could not resolve dependency:
npm error peer typescript@"4.0.x || 4.1.x || 4.2.x || 4.3.x || 4.4.x || 4.5.x" from typedoc@0.22.10
npm error node_modules/typedoc
npm error   dev typedoc@"^0.22.10" from the root project
npm error
npm error Conflicting peer dependency: typescript@4.5.5
npm error node_modules/typescript
npm error   peer typescript@"4.0.x || 4.1.x || 4.2.x || 4.3.x || 4.4.x || 4.5.x" from typedoc@0.22.10
npm error   node_modules/typedoc
npm error     dev typedoc@"^0.22.10" from the root project
npm error
npm error Fix the upstream dependency conflict, or retry
npm error this command with --force or --legacy-peer-deps
npm error to accept an incorrect (and potentially broken) dependency resolution.
```

Listing 7.7: Error reported upon peer dependency resolution failure during `ts-node` project build.

category could have their dependencies resolved successfully with older version of npm or with Yarn. It was found that 27 packages in this category do have a `yarn.lock` file in the VSC, indicating their developers likely use Yarn over npm.

7.2.5 Projects with invalid upstream lockfiles

The `npm uninstall` command was invoked during every project build to make sure an up-to-date lockfile is in place. Despite that, for two packages a lockfile was left behind that `npm ci` later reported as invalid due to being out of sync with project's `package.json`. One of these projects had a preexisting `package-lock.json` file and the other had a `yarn.lock` file[3].

7.2.6 Projects that expect network access to build

Projects in this category failed to build due to unsuccessful network request attempts other than the attempts mentioned at the beginning of 7.2.

The majority of build failures in this category occured when project's development dependency was trying to download a web browser binary for browser-based tests. Examples of other non-npm resources that projects tried to download were font files from

[3]npm also reads a `yarn.lock` when no other lockfile is present

Google Fonts and sources for automated native compilation of a library whose Node.js bindings package was being installed.

It can be stated that network accesses during npm project builds are commonly made to facilitate installation of architecture-specific software binaries, as these are inconvenient to distribute through the architecture-agnostic npm Registry.

7.2.7 Projects that require a build tool other than npm

Projects in this category are known to require either Yarn or pnpm to build. They could be classified with certainty because either

- their `package.json` files contained special URLs or package names that npm could not handle, or
- their build processes printed messages that explicitly informed the developer about the need to use a particular tool.

There are many more projects which likely rely on Yarn or pnpm but could not be classified here with certainty, see 7.2.4.

7.2.8 Projects with additional non-npm dependencies

Projects in this category need additional tools that are not installable through npm. Unlike projects mentioned in 7.2.6, these rely on the developer to install the additional tool.

Experiment logs indicated failures upon searching for Python executable and for configuration files of popular shells.

7.2.9 Projects with other build failures

Projects in this category failed to build due to reasons other than those discussed up to this point. Failures occured due to problems like missing modules, missing variable, and an operation hanging indefinitely.

7.2.10 Projects that could be built only when using upstream lockfiles

Projects in this category failed to build only after their upstream lockfiles were removed. After seemingly successfult dependency resolution, errors were raised during TypeScript compilation. The errors almost certainly resulted from newer versions of project's dependencies being used. This occured despite the use of Semantic Versioning and the respecting of dependency constraints declared by projects.

7.2.11 Projects built with both upstream and re-generated lockfiles

Packages in this category are considered to have been built successfully, because all npm command invocations from the first two builds finished with zero status. There is no guarantee that the packages built are fully functional. For example, some projects like `@testing-library/react` rely on an additional tool called semantic-release, which is not invoked as part of `npm run build`. That tool is responsible for analyzing project's change history and determining the right version number to be recorded in project's `package.json` file [Mar+nt]. When its use is omitted, the built package is reported as having a placeholder version, e.g., "0.0.0-semantically-released".

It is expected that a more polished and defect-free build process would often involve a dependency tree of several more or several less packages than in this experiment. Nonetheless, it was assumed that dependency sets found necessary for successful `npm install` and `npm build` invocations do represent the characteristics of the projects well enough.

Results presented through the rest of this chapter concern the dependency trees of projects from this very category.

7.3 Dependency trees after removals of dependencies and their conflicts

The sizes of the original dependency trees, produced in project builds with upstream lockfiles removed, are shown in Figure 7.3, together with the sizes of pruned trees. Each

pair of boxes represents the experiment at a different stage. The left box of each pair shows the average number of project dependencies where all distinct versions of the same package are counted. E.g., an npm package `foo` that occurs in a dependency tree three times as `foo@1.2.3`, `foo@2.1.0`, and again `foo@2.1.0` is counted as two. The right box of each pair shows the average number of dependencies with all versions of a single package counted as one. E.g., the `foo` introduced before is now counted as one. Standard deviation of the sample is additionally plotted over every box.

Individual projects' dependency tree sizes are plotted as circles over every box. 30 projects were found to also have their corresponding packages present in Debian Bookworm as of June 3rd, 2025. They are represented by filled, black circles. No clear relation between dependency tree sizes and presence in Debian can be seen at any stage of the experiment. Also consult Figure 7.5 for a variant of this plot that omits builds of packages which appear to be nonfunctional due to aggressive dependency elimination.

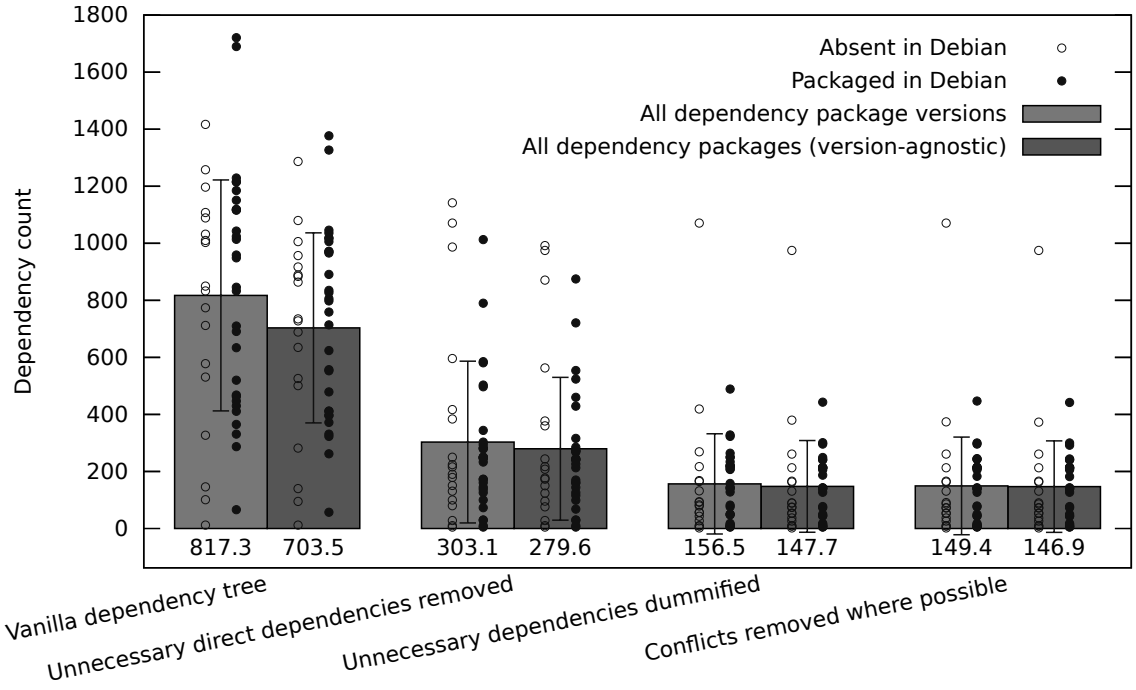

Figure 7.3: Dependency tree sizes of built npm projects.

7.4 Dependency conflict counts

Projects were further categorized by the number of dependency conflicts that could not be removed with the method used. The categories are visualized in Figure 7.4. Counts of all

projects and counts of projects having corresponding packages in Debian are shown. Also consult Figure 7.6 for a variant of this plot that omits builds of packages which appear to be nonfunctional due to aggressive dependency elimination.

As can be seen, most of the projects had few to no remaining dependency conflicts. Once again, there was no clear relation between the number of remaining dependency conflicts and presence in Debian. Given this distribution's norms, this suggest that the authors of Debian packaging likely managed to further remove some conflicts that could not be eliminated with the experiment's method. They possibly used more invasive methods like source code patching.

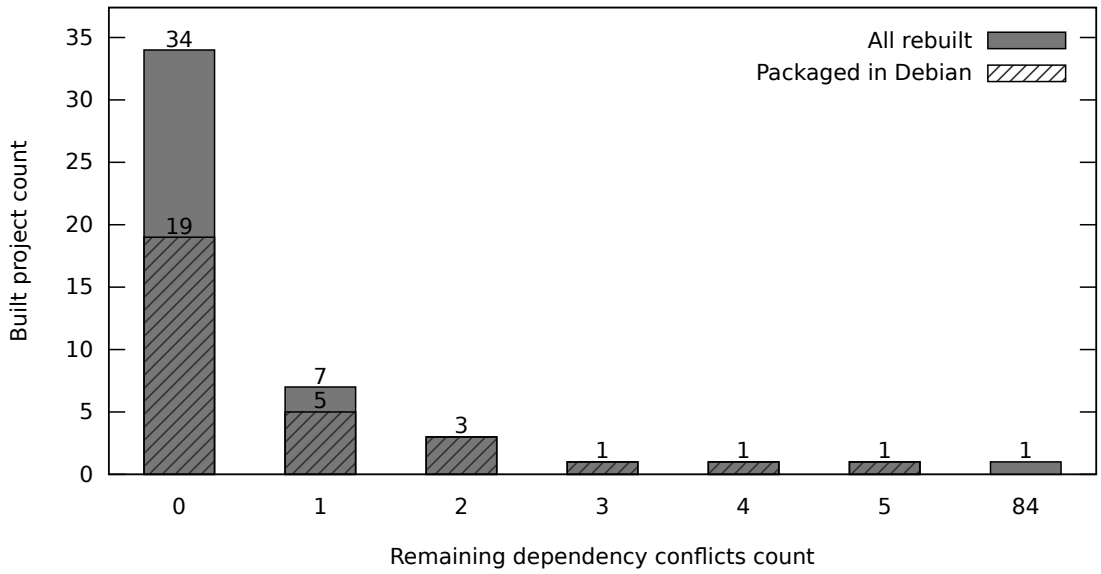

Figure 7.4: Counts of built projects with different numbers of unremovable dependency conflicts.

7.5 Differences in build outputs produced during the experiment

The repeated builds with upstream lockfiles removed had their produced package tarballs and generated `package-lock.json` files compared. The files produced on two build runs were found identical in case of every successfully built project[4]. This does not yet guarantee that the builds and dependency resolutions are reproducible. However, it does indicate

[4]npm authors should be credited for making `npm pack` produce tarballs without nondeterministic timestamps

that differences found after dependency removals, etc. are likely the effect of those alterations and not manifestations of builds nondeterminism.

It was found that 18 out of 48 successfully built projects had their package tarballs differ as the result of either dependency removals, dummifications, or conflict removals. All these cases were manually inspected. Two differences were found to be caused by reordered `package.json` entries and appear to be mere artifacts of this experiment's method of restoring project's original `package.json` file before `npm pack` invocation. Some of the more interesting cases are discussed below.

7.5.1 Use of a different dependency version

Several times an alteration of the build process allowed the npm's resolution algorithm to use a newer version of a dependency which then caused generation of different but still correct code. One example is the `concurrently` console application written in TypeScript. Package `typescript`, providing a TypeScript compiler, was specified as its direct development dependency. During the experiment this direct dependency was removed, but package `typescript` still made its way to the dependency tree due to being required by another dependency – `@hirez_io/observer-spy`. Its constraint on `typescript`'s version was looser than that present before, causing version 5.8.3 to be used instead of former 5.2.2. A sample of changes caused by the use of that newer TypeScript compiler is presented in Listing 7.8.

It is worth noting that the dependency `@hirez_io/observer-spy` – even if not necessary by itself – could not be eliminated with this experiment's method.

7.5.2 Inlining of a dependency

One similar case was that of built `axios` package, which – besides also showing generated code differences resulting from changed compiler/bundler version – had its dependency `proxy-from-env` treated differently, despite always occuring in the same version 1.1.0. Initially, `proxy-from-env` was specified as `axios`' runtime dependency and was merely referenced from the code being generated during build. When, as part of the experiment, `proxy-from-env` was removed as project's direct dependency, it remained present in the dependency tree due to being required by certain development dependency. `proxy-from-env` was therefore itself flagged as an indirect development dependency, which made the bundler treat it differently and inline it in `axios`' generated code.

```
| ├── package/dist/src/concurrently.d.ts
| | @@ -1,9 +1,7 @@
| | -/// <reference types="node" />
| | -/// <reference types="node" />
| |  import { Writable } from 'stream';
| |  import { CloseEvent, Command, CommandIdentifier, CommandInfo, KillProcess, SpawnCommand } from './
↪ command';
| |  import { SuccessCondition } from './completion-listener';
| |  import { FlowController } from './flow-control/flow-controller';
| |  import { Logger } from './logger';
| |  /**
| |   * A command that is to be passed into `concurrently()`.
| ├── package/dist/src/command-parser/expand-arguments.d.ts
| | @@ -5,14 +5,14 @@
| |   */
| |  export declare class ExpandArguments implements CommandParser {
| |      private readonly additionalArguments;
| |      constructor(additionalArguments: string[]);
| |      parse(commandInfo: CommandInfo): {
| |          command: string;
| |          name: string;
| | -        env?: Record<string, unknown> | undefined;
| | -        cwd?: string | undefined;
| | -        prefixColor?: string | undefined;
| | -        ipc?: number | undefined;
| | -        raw?: boolean | undefined;
| | +        env?: Record<string, unknown>;
| | +        cwd?: string;
| | +        prefixColor?: string;
| | +        ipc?: number;
| | +        raw?: boolean;
| |      };
| |  }
```

Listing 7.8: Excerpt from diffoscope's report of differences in built `concurrently` package tarballs.

7.5.3 Digest included in the generated package

If project's generated files include a hash of its dependency specifications or a similar derived value, it is an obvious source of difference in this experiment. One of such projects is `ts-jest`, which places a digest like `4ec902e59f1ac29ff410d624dcccf9b192920639` in a `.ts-jest-digest` file inside its package tarball.

7.5.4 Apparently disfunctional built packages

Several times a change to project's dependency tree did actually cause a significant change to the build output. In multiple cases, a removed package could not be found by a bundler tool called Rollup, which then treated it as an "external" dependency than need not be bundled. Rollup merely issued a warning about a reference to the now-absent code module and proceeded without it. An example of this can be seen in Listing 7.9 with an excerpt from the log of `rollup-plugin-typescript2` project build. `rollup-plugin-typescript2` specified package `object-hash` as a development dependency and had its code included in an

amalgamated script file generated by Rollup. After `object-hash` was removed, the invocation of `rollup-plugin-typescript2`'s build action still finished with zero status, with the generated amalgamated script file being smaller by the size of the missing dependency. If the `rollup-plugin-typescript2` package built this way were to be later used, its code would likely encounter an error when trying to import the `object-hash` module.

```
1
2  > rollup-plugin-typescript2@0.36.0 prebuild
3  > rimraf dist/*
4
5
6  > rollup-plugin-typescript2@0.36.0 build
7  > rimraf dist/* && rollup -c
8
9
10 src/index.ts → dist/rollup-plugin-typescript2.cjs.js, dist/rollup-plug
11 in-typescript2.es.js, build-self/dist/rollup-plugin-typescript2.es.js...
12 rpt2: typescript version: 5.8.3
13 rpt2: tslib version: 2.8.1
14 rpt2: rollup version: 2.79.2
15 rpt2: rollup-plugin-typescript2 version: 0.35.0
16 rpt2: ambient types changed, redoing all semantic diagnostics
17 rpt2: transpiling '/tmp/checkout/src/index.ts'
18 rpt2: transpiling '/tmp/checkout/src/context.ts'
19 rpt2: transpiling '/tmp/checkout/src/host.ts'
20 rpt2: transpiling '/tmp/checkout/src/tsproxy.ts'
21 rpt2: transpiling '/tmp/checkout/src/ioptions.ts'
22 rpt2: transpiling '/tmp/checkout/src/tscache.ts'
23 rpt2: transpiling '/tmp/checkout/src/rollingcache.ts'
24 rpt2: transpiling '/tmp/checkout/src/icache.ts'
25 rpt2: transpiling '/tmp/checkout/src/diagnostics.ts'
26 rpt2: transpiling '/tmp/checkout/src/diagnostics-format-host.ts'
27 rpt2: transpiling '/tmp/checkout/src/parse-tsconfig.ts'
28 rpt2: transpiling '/tmp/checkout/src/get-options-overrides.ts'
29 rpt2: transpiling '/tmp/checkout/src/tslib.ts'
30 rpt2: rolling caches
31 (!) Unresolved dependencies
32 https://rollupjs.org/guide/en/#warning-treating-module-as-external-dependency
33 object-hash (imported by src/tscache.ts)
34 created dist/rollup-plugin-typescript2.cjs.js, dist/rollup-plugin-typescript2.es.js, build-self/dist/rollup-
↪  plugin-typescript2.es.js in 1m 0.7s
```

Listing 7.9: The output of `npm run build` invocation with a missing dependency reported by Rollup.

A single interesting case was that of `@testing-library/user-event` project and the `package.json` file generated for its package tarballs. Several – seemingly vital – `package.json` keys were no longer present after project's `typescript` dependency was removed. The change caused by `typescript`'s removal is shown in Listing 7.10).

In some cases, as subsequent dependencies of a project were eliminated, a bundler tool combined the dependencies in a different order, making amalgamated scripts difficult to compare with diff-like tools. There were also cases where the size of individual generated files would increase by an order of magniture or even go up and down during a series of builds. One suspected reason for increasing amalgamated script size – besides

```
| ├── package/package.json
| | ├── Pretty-printed
| | | @@ -3,6 +3,7 @@
| | |       "bugs": {
| | |           "url": "https://github.com/testing-library/user-event/issues"
| | |       },
| | | +     "dependencies": {},
| | |       "description": "Fire events the same way the user does",
| | |       "devDependencies": {
| | |           "@esbuild-plugins/node-modules-polyfill": "^0.2.2",
| | | @@ -40,23 +41,6 @@
| | |           "node": ">=12",
| | |           "npm": ">=6"
| | |       },
| | | -     "exports": {
| | | -         ".": {
| | | -             "default": "./dist/esm/index.js",
| | | -             "require": "./dist/cjs/index.js",
| | | -             "types": "./dist/types/index.d.ts"
| | | -         },
| | | -         "./dist/cjs/*.js": {
| | | -             "default": "./dist/cjs/*.js",
| | | -             "import": "./dist/esm/*.js",
| | | -             "types": "./dist/types/*.d.ts"
| | | -         },
| | | -         "./dist/esm/*.js": {
| | | -             "default": "./dist/esm/*.js",
| | | -             "require": "./dist/cjs/*.js",
| | | -             "types": "./dist/types/*.d.ts"
| | | -         }
| | | -     },
| | |       "files": [
| | |           "dist"
| | |       ],
| | | @@ -68,8 +52,6 @@
| | |           "testing"
| | |       ],
| | |       "license": "MIT",
| | | -     "main": "./dist/cjs/index.js",
| | | -     "module": "./dist/esm/index.js",
| | |       "name": "@testing-library/user-event",
| | |       "peerDependencies": {
| | |           "@testing-library/dom": ">=7.21.4"
```

Listing 7.10: Excerpt from diffoscope's report of differences in `package.json` files inside built `@testing-library/user-event` package tarballs.

the one discussed in 7.5.2 – is polyfilling. It is the action of replacing newer JavaScript language constructs used by programmers with code that is compatible with older language runtimes. An older version of a build tool would typically aim to support more legacy runtimes, applying more polyfills and increasing the produced script sizes as a result. Nonetheless, for the purpose of this experiment, whenever the nature and effect of changes in a build output were unclear, the package was considered one of the total of eight disfunctional packages.

7.5.5 Updated statistics

We are interested in the relation between project's dependency tree characteristics and its packagability for software distributions. The statistics presented in 7.3 and 7.4 included

eight projects with assumed disfunctionalities introduced by this very experiment. Three of these do have corresponding Debian packages. As these eight cases could make our results deceptive, the statistics are now presented again, with problematic projects not taken into account. Dependency tree sizes at various stages of the experiment are presented in Figure 7.5. Projects' categorization by the number of remaining dependency conflicts is visualized in Figure 7.6.

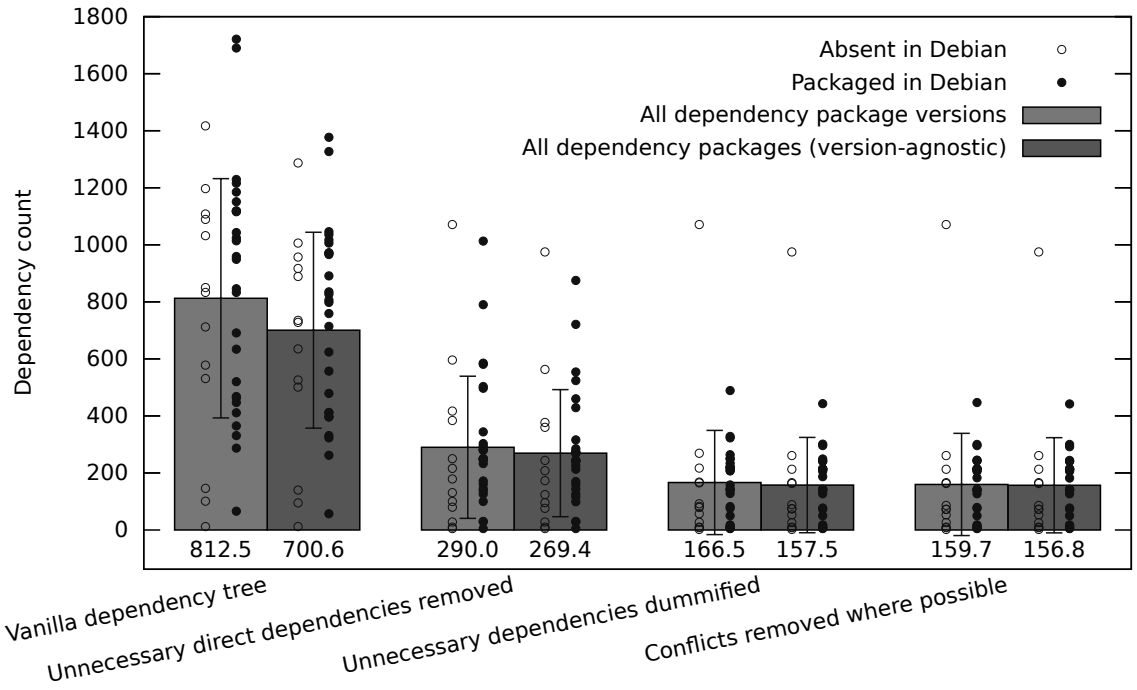

Figure 7.5: Dependency tree sizes of built npm projects. Packages which appear to be nonfunctional due to aggressive dependency elimination are not included.

As one can see, even these "cleaned" results show no relation between project's dependency tree sizes and its Debian presence.

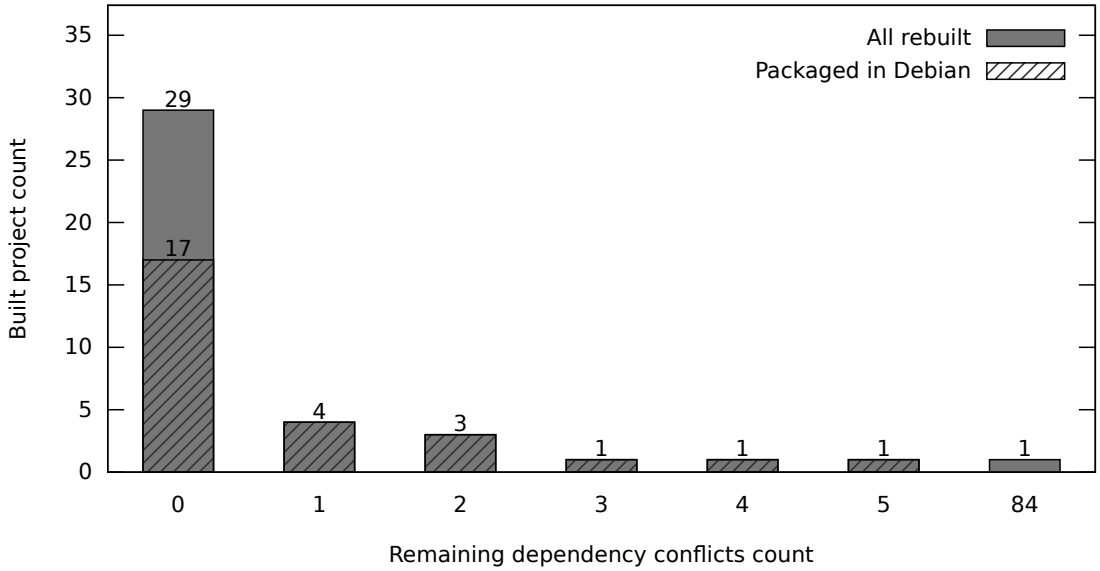

Figure 7.6: Counts of built projects with different numbers of unremovable dependency conflicts. Packages which appear to be nonfunctional due to aggressive dependency elimination are not included.

8. Conclusions

The results of conducted experiment allow the questions stated at the beginning of 7 to be answered.

8.1 Naming the main hindrance to packaging the npm ecosystem

We expected that huge dependency trees and presence of conflicting dependencies are the major obstacles to packaging npm projects into reproducibility-focused software distributions. The experiment results show the contrary. If this hypothesis were true, we would see npm projects with more complex dependency trees less frequently packaged into Debian – but there is no such relation.

What are then the most likely reasons for relatively small number of software from the npm ecosystem in Debian and GNU Guix? For the latter, the challange of bootstrapping several popular, self-depending build tools appears relevant. Five of these were mentioned in 4.5.1. As of June 3rd, 2025, four of them – `typescript`, `@babel/core`, `rollup`, and `gulp` – are present in Debian, albeit under different names. The Debian packages of all four were also found to depend on themselves to build. Since GNU Guix' policies make it more difficult to add self-depending packages to the distribution, this explains the drastically different coverage of the npm ecosystem by these system software distributions.

Aside from that, the incompatibility of JavaScript developers' workflows with system software distribution packages – as highlighted in 4.6 – should be considered the major issue.

Kraków, 2025

8.2 Developer-supplied lockfile being infrequently necessary

As found, only two tested projects could be built with an upsteam lockfile but failed when it was re-generated. Meanwhile, 48 packages were built successfully in both ways. This is consistent with the expectations. Repetition of the dependency resolution is not a likely source of build failures.

8.3 Indispensibility of direct and indirect npm build dependencies

It was found that a non-negligible subset of both direct and indrect npm project dependencies is unnecessary for a successful build. The effect of removal of unnecessary direct dependencies can be comprehended by comparing the leftmost two pairs of boxes in Figure 7.5. There is on average an almost triple reduction in dependency tree sizes, although with a huge variance. The average number of direct dependencies shrank from 31.6 with sample standard deviation of 16.4 to 7.3 with sample standard deviation of 4.9.

Comparing the second and third pair of boxes in Figure 7.5 shows that almost a half of projects' remaining indirect dependencies is also not necessary during build, again with a huge variance. The experiment's automated method of determining the unnecessary dependencies was not perfect, as shown in 7.5.1, but sufficient to allow a conclusion that during packaging, the elimination of both kinds of dependencies can be worth attempting.

8.4 Typical dependency tree sizes of npm projects

The average sizes of built projects' dependency trees ranged from 12 to 1721 with sample standard deviation of 419.5, as shown in Figure 7.5. With the experiment's method, it was possible to reduce the tree sizes by about a ratio of five on average.

The final dependecy tree sizes of about 160 are not drastically higher than those in other software ecosystems. For example, as of June 3rd, 2025, Debian package `python-xrt` was found to be built in an environment with 180 installed packages named `python-*`, as

evidenced in its `.buildinfo` file.

It is worth noting that the numbers discussed here and in the following section might be representative of only the more complex npm packages. As explained in 7.2.3, there can be many popular npm projects like `semver` that do not require an actual `build` action, likely have fewer declared dependencies, and likely only require the npm tool to create the package tarball.

8.5 Frequency of dependency conflicts in npm projects

Among the npm projects built successfully, only one had no conflicting dependencies in its original tree. This shows that dependency conflicts are indeed a normal and accepted thing in the npm ecosystem.

In the original dependency trees, the count of dependencies in conflict averaged at 86.7 and sample standard deviation was 57.5. In case of more than half of the projects the conflicts were completely eliminated. Several cases of unremovable conflicts remained, as can be sees in Figure 7.6. However, this part of the results should not be considered entirely representative of the real state of affairs, as explained in 7.4. It is expected that through manual work the build process of many more npm packages can be made free of dependency conflicts.

8.6 Package disfunctionality caused by dependency tree reduction

As witnessed in 7.5.4, there is a non-negligible number of cases where forced removal of direct or indirect dependencies causes built package to lack important pieces of code or have other deficiencies. Many of these problems were caused by Rollup bundler's liberal treatment of missing code modules and could be automatically detected, for example by searching the build logs for specific strings.

Nevertheless, the risk of building disfunctional packages appears relatively high, which is not acceptable if this experiment's method were to be used for preparation of package recipes in a reproducibility-oriented software distribution. Since in more than half of all cases the diffoscope's reports on built package differences were found comprehensible, it

is advisable to manually investigate dependency removals whose effects are unclear.

Additionally, the method itself proved to have a weakness of allowing a removed direct dependency to still appear as an indirect dependency, as shown in 7.5.1 and 7.5.2. Although this does not appear to have lead to built packages' disfunctionalities during the experiment, it is a space for improvement. One simple solution would be to eliminate direct dependencies through dummification, as it was already done with indirect ones.

8.7 Relevance of npm package distribution tags for successful dependency resolution

The dependency resolution failures described in 7.2.4 were all manually analyzed and none was found to be caused by a dependency specification referencing a distribution tag omitted in the experiment. Four of the built projects were found to have a direct or indirect dependency specified by the `latest` tag. Among others, `typescript` – the most popular npm package according to the ranking in 5.3 – requires its direct development dependency `@types/node` to be in version tagged `latest`.

Concluding, the special `latest` tag should be present in npm dependency metadata to avoid needless dependency resolution failures. Fortunately, it can usually be derived from the available package versions. All other tags can in the vast majority of cases be omitted from the dependency resolution matadata, removing the need to rely on external, mutable npm distribution tags information.

8.8 Prototype for npm dependency resolution under Paradigm 3

The experiment's environment resembled that proposed in 6.4 for Paradigm 3 for hermeticity and reproducibility. The approach with a host-side service performing requests on behalf of the isolated guest was indeed workable. The experiment also showed that this prototype could benefit from added ability to provide the guested npm process with dependency package files hosted on different sites than just the npm Registry. This would require additional work to have npm's package tarball requests reach the locally-running

service. A solution could involve configuring npm to use a TLS-enabled HTTP proxy in the likes of mitmproxy [Cor+nt]. While burdensome, it should be workable.

At the same time, obtained results did not contain the expected indicators that Paradigm 1 – and Paradigm 2 with flat input metadata of the dependency resolution step – is insufficient in practice for handling the complex npm ecosystem. This means that new paradigms, as proposed in this work, are not necessary for further progress in the field of reproducible software builds. However, paradigms 3 and 4 can still prove useful in addressing the bootstrappability and developer usefulness issues named in 8.1.

9. Summary

Throughout the course of this work it was found that software industry shows some modest interest in reproducible builds. Practical difficulties in applying reproducibility to software projects hinder popularisation of this security measure. Software developed around popular technologies must become easier to rebuild hermetically and to test for reproducibility. This will allow reproducible builds to be more broadly recommended and mandated.

Even though the concept of end user verification of build reproducibility offers increase in security confidence, years after 2018 saw little progress in its application. Due to the issues of rebuilder lag and occasional build nondeterminism, continuous tests performed independently of end users should be considered a more practically viable security measure.

Even when build reproducibility is aimed and tested for, software distributions' approaches differ. Metadata used to reproduce Debian packages was found insufficient to also reproducibly verify the results of dependency resolutions that package builds relied on. This is not a severe vulnerability. However, it motivates increased interest in purely functional package managers, whose design excludes the possibility of such "verification hole" occuring.

Contrary to intuition, traditional software packaging scheme of Debian can be applicable even to npm software with complex dependency graphs. One could still attempt to utilize suggested Paradigm 3 or 4 to replace existing approaches of Debian and GNU Guix. However, such efforts would offer no certain benefit.

Although npm packages tend to have large dependency tree sizes and conflicting dependencies, this is not the ultimate reason for almost zero coverage of the npm ecosystem by GNU Guix. Instead, the issues of package bootstrappability appear to be the determining factor. These, however, are practically solvable.

The packages in reproducible software distributions must be bridged with developers'

preferred workflows. If this does not happen, the distributions will not only be slow in including software from the npm Registry and similar repositories. The software already in those distributions will fail to bring the security benefits that it could.

As long speculated, much of declared dependencies of a typical npm project are not needed to build it. It was found that many indirect dependencies are also unnecessary. Their omission is crucial both to simplify packaging into software distributions and to reduce the software supply chain attack surface.

10. Future work

During npm project builds analysis it was found that projects exist which require no code transformations during build. One such case is described in detail in 7.2.3. If identified, projects from this category could be packaged hermetically and reproducibly with minimal effort. Automated identification of such projects could be a future research goal.

After the experiment, bootstrappability was named a likely major reason for small coverage of the npm ecosystem by GNU Guix. The finding of viable and optimal bootstrap chains of npm packages could therefore be the subject of another research.

Paradigms 3 and 4 for hermeticity and reproducibility were proposed to address the issue of incomprehensibly complex dependency relations between npm packages. It was found that in the context of reproducible builds, the issue is resolvable even without employing these paradigms. However, it is yet to be checked – possibly in a new work – whether these paradigms can actually make software packaging process less labor-intensive and therefore more efficient.

This work touches the topic of securing the inputs of software builds. The applicability of methods like the suggested canaries could be further investigated.

A subset of tested projects did not have their VCS revisions tagged in the expected way. The tagging habits of developers and means of automated identification of VCS revisions corresponding to releases could be researched. A possible approach to solving the problem of missing VCS tags could involve comparing commit dates with package version release dates. If devised, a successful method would benefit VCS-based software packaging, which appears desirable in the light of the XZ backdoor.

npm developers are not incentivized make their software easily bootstrappable. Proof of concept of Ken Thompson's Trusting Trust attack [Tho84] could be presented for one or more popular npm packages. It could help raise awareness of the supply chain issues and make the community interested in rebuildability and ultimately bootstrappability. The PoC could be a self-implanting backdoor in one of the self-depending builds tool.

Kraków, 2025

Bibliography

[Abd+20] Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. "On the impact of using trivial packages: an empirical case study on npm and PyPI". In: *Empirical Software Engineering* 25.2 (Mar. 2020), pp. 1168–1204. ISSN: 1573-7616. DOI: 10.1007/s10664-019-09792-9. URL: https://rabeabdalkareem.github.io/files/12-abdelkareem_emse2020.pdf (visited on July 3, 2025).

[Adr+25] Diglio Adrian et al. *Supply-chain Levels for Software Artifacts*. Open Source Security Foundation. 2025. URL: https://slsa.dev/spec/v1.1/ (visited on July 2, 2025).

[BCR15] Scott Bauer, Pascal Cuoq, and John Regehr. "Deniable Backdoors Using Compiler Bugs". In: *International Journal of PoC ‖ GTFO* 8.3 (2015). URL: https://www.alchemistowl.org/pocorgtfo/pocorgtfo08.pdf.

[Boj20] Tao Bojlén. *A web of trust for npm*. 2020. URL: https://btao.org/posts/2020-10-02-npm-trust/ (visited on July 3, 2025).

[Cor+nt] Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. *mitmproxy: A free and open source interactive HTTPS proxy*. [Version 12.0]. 2010–present. URL: https://mitmproxy.org/.

[Cou+24] Ludovic Courtès, Timothy Sample, Simon Tournier, and Stefano Zacchiroli. "Source Code Archiving to the Rescue of Reproducible Deployment." In: *CoRR* abs/2405.15516 (2024). URL: http://dblp.uni-trier.de/db/journals/corr/corr2405.html#abs-2405-15516.

[Cou13] Ludovic Courtès. "Functional Package Management with Guix." In: *ELS*. Ed. by Christian Queinnec, and Manuel Serrano. ELSAA, 2013, pp. 4–14. URL: http://dblp.uni-trier.de/db/conf/els/els2013.html#Courtes13.

[Cou15] Ludovic Courtès. *GNU Guix 0.9.0 released*. 2015. URL: https://savannah.gnu.org/forum/forum.php?forum_id=8398 (visited on July 2, 2025).

[Cou22] Ludovic Courtès. "Building a Secure Software Supply Chain with GNU Guix". In: *The Art, Science, and Engineering of Programming* 7.1 (2022). DOI: https://doi.org/10.22152/programming-journal.org/2023/7/1. eprint: 2206.14606 (cs.SE). URL: https://programming-journal.org/2023/7/1/.

[DHV] Joshua Drexel, Esther Hänggi, and Iyán Méndez Veiga. *Reproducible Builds and Insights from an Independent Verifier for Arch Linux*. DOI: https://doi.org/10.18420/sicherheit2024_016. arXiv: 2505.21642 [cs.CR].

[Dig+22] Adrian Diglio, Jay White, Jasmine Wang, Tom Bedford, Christopher Robinson, and David A. Wheeler. *Secure Supply Chain Consumption Framework*. Open Source Security Foundation. 2022. URL: https://openssf.org/projects/s2c2f/ (visited on July 2, 2025).

[Dol06] Eelco Dolstra. "The purely functional software deployment model." PhD thesis. Utrecht University, Netherlands, 2006. URL: https://edolstra.github.io/pubs/phd-thesis.pdf.

[Gos20] Pronnoy Goswami. "Investigating the Reproducbility of NPM packages". Master's thesis. Virginia Polytechnic Institute and State University, 2020. URL: https://vtechworks.lib.vt.edu/server/api/core/bitstreams/3ef5408d-8617-4993-ac7e-d171a13dfa22/content.

[Kas19] Andrei Kashcha. *npm rank*. 2019. URL: https://gist.github.com/anvaka/8e8fa57c7ee1350e3491 (visited on July 3, 2025).

[Lak25] Ravie Lakshmanan. *Over 70 Malicious npm and VS Code Packages Found Stealing Data and Crypto*. The Hacker News. 2025. URL: https://thehackernews.com/2025/05/over-70-malicious-npm-and-vs-code.html (visited on July 3, 2025).

[Lam+] Chris Lamb et al. *SOURCE_DATE_EPOCH*. Reproducible Builds. URL: https://reproducible-builds.org/docs/source-date-epoch/ (visited on July 8, 2025).

[Lem15] Christine Lemmer-Webber. *Let's Package jQuery: A Javascript Packaging Dystopian Novella*. 2015. URL: https://dustycloud.org/blog/javascript-packaging-dystopia/ (visited on July 3, 2025).

[Lev+25] Holger Levsen, Hideki Yamane, Lucas Nussbaum, Andreas Barth, Raphaël Hertzog, Adam Di Carlo, and Christian Schwarz. *Debian Developer's Reference*. 13.20. Debian project. 2025. URL: https://www.debian.org/doc/manuals/developers-reference/index.en.html (visited on July 3, 2025).

[Lin+24] Mario Lins, René Mayrhofer, Michael Roland, Daniel Hofer, and Martin Schwaighofer. "On the critical path to implant backdoors and the effectiveness of potential mitigation techniques: Early learnings from XZ." In: *CoRR* abs/2404.08987 (2024). URL: https://dblp.uni-trier.de/db/journals/corr/corr2404.html#abs-2404-08987.

[Lkcnt] Morten Linderud, kpcyrd, and contributors. *archlinux-repro: A tool for users to verify packages distributed by Arch Linux*. 2017–present. URL: https://github.com/archlinux/archlinux-repro/ (visited on July 17, 2025).

[LZ21] Chris Lamb, and Stefano Zacchiroli. "Reproducible Builds: Increasing the Integrity of Software Supply Chains". In: *CoRR* abs/2104.06020 (2021). arXiv: 2104.06020. URL: https://arxiv.org/abs/2104.06020.

[Mar+nt] Gregor Martynus, Pierre Vanduynslager, Matt Travi, Stephan Bönnemann, Rolf Erik Lekang, Johannes Jörg Schmidt, Finn Pauls, and Christoph Witzko. *semantic-release: Fully automated version management and package publishing*. [Version 24.2.6]. 2015–present. URL: https://www.npmjs.com/package/semantic-release (visited on July 3, 2025).

[Mil18] Danny Milosavljevic. *Bootstrapping Rust*. 2018. URL: https://guix.gnu.org/blog/2018/bootstrapping-rust/ (visited on July 3, 2025).

[Mun23] Phil Muncaster. *Hundreds of Malicious Packages Found in npm Registry*. Infosecurity Magazine. 2023. URL: https://www.infosecurity-magazine.com/news/hundreds-malicious-packages-npm/ (visited on July 3, 2025).

[Nap25] Ernestas Naprys. *Dozens of malicious packages on NPM collect host and network data*. Cybernews. 2025. URL: https://cybernews.com/security/

node-developers-targeted-by-malware-in-npm-packages/ (visited on July 3, 2025).

[Nic22] Shaun Nichols. *More than 1,000 malware packages found in NPM repository*. TechTarget. 2022. URL: https://www.techtarget.com/searchsecurity/news/252512799/More-than-1000-malware-packages-found-in-NPM-repository (visited on July 3, 2025).

[NOC22] NSA, ODNI, and CISA. *Securing the Software Supply Chain: Recommended Practices Guide for Developers*. Executive Order (EO) 14028. Enduring Security Framework. Aug. 2022. URL: https://www.cisa.gov/sites/default/files/publications/ESF_SECURING_THE_SOFTWARE_SUPPLY_CHAIN_DEVELOPERS.PDF (visited on July 3, 2025).

[NSA24] NSA. *Recommendations for SBOM Management*. Executive Order (EO) 14028. 2024. URL: https://media.defense.gov/2023/Dec/14/2003359097/-1/-1/0/CSI-SCRM-SBOM-MANAGEMENT.PDF (visited on July 3, 2025).

[Pre13] Tom Preston-Werner. *Semantic Versioning*. web. 2013. URL: http://semver.org/ (visited on July 3, 2025).

[Ray01] Eric S. Raymond. *The cathedral and the bazaar: musings on Linux and open source by an accidental revolutionary*. eng. 2., überarb. und erw. A. With a foreword by Bob Young. Beijing; Cambridge; Farnham; Köln; Paris; Sebastopol; Taip: O'Reilly Media, 2001, p. 241. ISBN: 0-596-00108-8. URL: http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/.

[RBCT] *Continuous tests*. Reproducible Builds. URL: https://reproducible-builds.org/ (visited on July 17, 2025).

[Rod+22] Josip Rodin, Osamu Aoki, Craig Small, and Raphaël Hertzog. *Debian New Maintainers' Guide*. Debian project. 2022. URL: https://www.debian.org/doc/manuals/maint-guide/index.en.html (visited on July 2, 2025).

[SB21] Lindsay Sterle, and Suman Bhunia. "On SolarWinds Orion Platform Security Breach." In: *SmartWorld/SCALCOM/UIC/ATC/IOP/SCI*. IEEE, 2021, pp. 636–641. ISBN: 978-1-6654-1236-0. URL: https://dblp.uni-trier.de/db/conf/uic/uic2021.html#SterleB21.

[Spr+20] Steve Springett, Dave Russo, Fick Garret, Herz JC, Scott John, Symons Mark, Nallapareddy Pruthvi, and Garcia Bryan. *Software Component Verification Standard*. The OWASP Foundation. 2020. URL: https://owasp.org/www-project-software-component-verification-standard/ (visited on July 2, 2025).

[Tho84] Ken Thompson. "Reflections on trusting trust". In: *Communications of the ACM* 27.8 (Aug. 1984), pp. 761–763. URL: https://www.cs.cmu.edu/~rdriley/487/papers/Thompson_1984_ReflectionsonTrustingTrust.pdf.

[Tor+19] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. "in-toto: Providing farm-to-table guarantees for bits and bytes." In: *USENIX Security Symposium*. Ed. by Nadia Heninger, and Patrick Traynor. USENIX Association, 2019, pp. 1393–1410. URL: https://dblp.uni-trier.de/db/conf/uss/uss2019.html#Torres-AriasAKC19.

[Tou25] Bill Toulas. *Dozens of malicious packages on NPM collect host and network data*. BleepingComputer. 2025. URL: https://www.bleepingcomputer.com/news/security/dozens-of-malicious-packages-on-npm-collect-host-and-network-data/ (visited on July 3, 2025).

[Veg+21] Andres Vega et al. *Software Supply Chain Security Best Practices*. Cloud Native Computing Foundation. 2021. URL: https://project.linuxfoundation.org/hubfs/CNCF_SSCP_v1.pdf (visited on July 2, 2025).

[W3JL] *Usage statistics and market shares of JavaScript libraries*. W3Techs - World Wide Web Technology Surveys. URL: https://w3techs.com/technologies/overview/javascript_library (visited on July 3, 2025).

[Whe09] David A. Wheeler. "Fully Countering Trusting Trust through Diverse Double-Compiling." PhD thesis. George Mason University, Fairfax, Virginia, USA, 2009. URL: https://dwheeler.com/trusting-trust/.

[zam22] zamfofex. *Re: bringing npm packages to Guix*. Message to a GNU Guix mailing list. 2022. URL: https://lists.gnu.org/archive/html/guix-devel/2022-11/msg00234.html (visited on July 3, 2025).