Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science

im Studiengang Informatik
der Fakultät Informatik und Medien
der Hochschule für Technik, Wirtschaft und Kultur Leipzig

Secure Nix Expression Updates

vorgelegt von

Finn Landweber
finn@landweber.xyz

Leipzig, den 23.07.2024

Abstract

Projects and individual users often struggle to keep track of their deployed software and update vulnerable versions quickly. Nix, an increasingly popular package manager, provides a rigorous approach to dependency management and transparency and could be used to improve this situation significantly. However, updates to its build instructions are not cryptographically secured and thus give way to machine-in-the-middle attacks. This thesis demonstrates that instruction updates can be protected from these kinds of attacks with minimal changes to the update mechanism. It takes Git as the basis for distributing versioned and signed Nix code and explores multiple ways in which the origins of downloaded instructions can be verified automatically. The work contributes a structured analysis of Nix instruction update security based in literature. From there, it derives novel interfaces from Nix to two preexisting Git signature verification solutions and presents a new tool tailored to the needs of the Nix ecosystem. Although not all attacks on Nix expression updates can be mitigated by the suggested tools, they can provide a practical security gain for Nix users. The findings may help conceptualize a path towards a higher security standard for Nix deployments.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich diese Bachelorarbeit selbstständig ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen sind als solche einzeln kenntlich gemacht. Diese Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht worden. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Leipzig, den 23.07.2024 .....

Contents

Chapter 1

Introduction

IT security encompasses a variety of specialized fields, each focusing on the protection of different aspects of digital infrastructure and data. For example, while application security aims to protect software from vulnerabilites through secure coding practices, network security safeguards data as it travels between computers. Supply-chain security addresses the protection of hardware and software throughout their lifecycle, from production to deployment, and will be the focus of this work.

Although each field plays a critical role in creating secure systems, supply-chain security is moving into the spotlight[36]. The challenge is to not only find mechanisms that *theoretically could* prevent compromises, but also to integrate them well enough into systems to provide protections *in practice*. Nix, as a modern package manager with conceptual clarity, may be well-equipped to rise to this challenge[42]. It allows users to audit the build process of programs and offers robust protection against common attacks, such as compromised artifact download servers.

Users write Nix expressions to build and deploy their programs and even operating systems[51]. To do so, they commonly heavily rely on other developers' work. This is especially true for Nixpkgs, which is a collection of expressions, that is used as the basis for most deployments. As with most software, Nix expressions are commonly distributed via download servers by their developers. Users can reference the expressions' URIs in their own Nix code and cause them to be downloaded on execution. Typically Nix will ensure to always download the same version of an expression. However, often users need to update the expression version to receive bugfixes in their dependencies. This in turn has the consequence that successful attacks on an expression's host server could also easily compromise their consumers. So while it is common practice to trust other developers not to provide malicious instructions, Nix users are also required to trust the distribution infrastructure. It would be trivial for an attacker controlling the download server or the respective domain to provide expressions that result in insecure or malicious programs.

When compared to other package managers' practices, this is a significant shortfall behind industry standards[12] and presents an obstacle for adoption in more security-sensitive contexts[52]. The past has shown, that even security-aware projects are vulnerable to attacks on their distribution mechanisms[29, 41]. With supply-chain hardening efforts for Nix on their way[28] and moves towards even more community-owned infrastructure on the horizon[1], it is important to secure Nix deployments against them.

This thesis is concerned with ensuring that the expressions a Nix user downloads are the ones they were intended to obtain by the developers in the case of an machine-in-the-middle attack. It therefore focuses on integrity of the expressions and is not primarily concerned with their availability nor their confidentiality. I propose to empower the user to verify the origin of a given Nix expression. This can be done by enabeling authors to cryptographically sign them and automatically checking those signatures against pre-known keys. It would thereby be possible to exchange expressions via distrusted infrastructure and reduce the attack surface of updates significantly.

[1] https://lix.systems

I approach this by using the Version Control System Git as a basis for authenticated expressions, as it is already widespread for Nix expression distribution and allows signing content. In this work, I introduce three distinct ways to allow users to authenticate the authors of Nix expressions when including them in their own code. As the simplest solution I propose a Nix interface to Git's own signature checking, which has already been included as an experimental feature. To prohibit more attacks as well as enabling signing for larger projects, such as Nixpkgs, I also derived a novel verification tool, Git-Verify. It is based on Git's signing capabilities but can realize more fine-grained permissions, provides key-management and puts an emphasis on simplicity and usability. Thirdly I distill a way to include the verification process of the Git development security tool Gittuf into Nix. Additionally, I present a modification to the locking system Niv that prevents malicious actors from downgrading the deployed signed expressions. I find that, whilst not all attacks on update integrity can be mitigated by these measures, the tools provide an immense improvement over the current situation and could help make Nix deployments more secure.

The remainder of this work is structured as follows: in chapter 2 I will provide some background on update security and further introduce Nix and Git. Chapter 3 will then define the threat model and describe viable attacks on Nix's distribution mechanisms. In chapter 4 I will introduce my solutions to the problem, which I will then discuss in chapter 5. Chapter 6 will conclude the thesis.

Chapter 2

Foundations

To follow the rest of this work, the reader requires a fundamental understanding of the tools involved. This chapter first attempts to provide an overview about the aspects of Git (section 2.1) and Nix (section 2.2). Readers already familiar with the tools may choose to skip those sections without loss. Section 2.3 provides context on prior work in update security research, which will be built upon in later chapters.

2.1 Git

Git[1] is a Version Control System (VCS) and is commonly used for developing code and other text-based projects collaboratively and asynchronously. This section will introduce basic concepts of Git, such as commits and tags, along with their formal denotation.

Git projects are organized in *repositories*, folders containing files and additional metadata, including their history, contributors and settings. A project's history can be described as a directed acyclic graph (DAG), where each node, known as a *commit*, is a state of the project with additional metadata[1:1]. Figure 2.1 shows such a commit graph with two diverging development histories. I will denote the set of all commits of a given project as $Commits$.

Parents are other commits, which a given commit is based on. The changes from a commit's parents to its own state are the work that is represented by the commit[1:9]. The set of parents of a commit $c \in Commits$ will be denoted as $Parents_c$. For the example in fig. 2.1 is $Parents_{c4} = \{c3\}$. All commits in the transitive closure of the parent relationship are a commit's ancestors.

A commit's *tree* references the files, that are contained in its state of the project[1:9]. It can be viewed as a snapshot of the respective file tree. For a commit $c \in Commits$ its tree shall be denoted as $Tree_c$.

A commit contains the following metadata:

- the commit's parents
- the commit's tree
- name and email address of the author, who is typically the person that came up with the change, and the time they created it
- name and email address of the committer, who is typically the person adding the change to the repository, and the time they committed it
- an arbitrary text as commit message

Commits are referred to by a hash sum, that is calculated over all attributes. Changing any attribute would also change the result of the calculation, which means no part of the commit can be altered without changing the name of the commit. This principle is called *content addressable* [15:10.2]. Git's implementation relies on the SHA1 hashing algorithm. Even though a viable attack on SHA1 for a subclass of files was found[58], Git's usage of it is still considered secure[31].

[1] https://git-scm.com

In addition to the previously mentioned metadata, a commit may also contain a cryptographic signature. This attribute allows a developer to sign all other metadata attributes (parents, tree, etc.) using their private key[15:7.4]. Other Git users can then check the signature with a known public key to verify, that the creator of the commit was in possession of the corresponding private key. The Git command line implements support for signing commits with OpenPGP[10], SSH[53] and x509[16] keys as well as user-defined tools[32].

Commits without parents are called *initial commits*. In fig. 2.1 $c1$ is the only initial commit.

Commits with multiple parents are called *merge commits*, as they bring together two diverging paths of the development history. In fig. 2.1 is $c5$ a merge commit because $|Parents_{c5}| > 1$. Sometimes Git can produce the merged tree from a given set of commits automatically (without requiring conflict resolution from the user). This is for example the case, if no file was changed in multiple diverging paths. A merge commit produced in such a way will be called an *auto-merge*. An auto-merged tree of commits $Cs \subseteq Commits$ is denoted as $automerge(Cs)$.

Commits can be referred to by *references*. References are not content addressable, but instead a way to link to a commit in a human-readable and mutable form. Branches and lightweight tags are forms of references, which are then interpreted accordingly by Git. Figure 2.1 displays the two branch heads `main` and `feature` as well as the lightweight tag `v1.0`.

Tags are objects that also refer to a commit by its hash. Similar to commits, a tag contains information on when and by whom it was created and additional free text[15:2.6]. Tags can be signed as well[15:7.4].

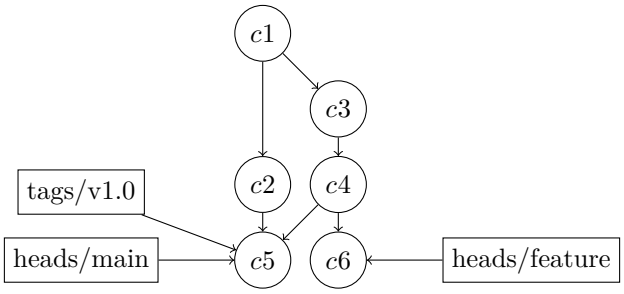


Figure 2.1: Commit graph

2.2 Nix

Nix[2][26] is a high-level build tool that is derived from the functional programming paradigm. Specifically, Nix models each build process as a mathematical function. In this model, the build's dependencies describe a function's arguments and the build artifact can be deterministically calculated from them through the build instructions. Nix can be used to unambiguously define build steps of artifacts, such as software binaries or this PDF, and therefore facilitate reproducibility. Amongst others, this is useful for enabling reproducible research in different scientific disciplines[3, 8, 9, 21, 38] and provide better testing[7].

This section will define the aspects of the tool, that are needed for understanding the thesis. It will start with the basics of expressions, components and reproducibility, continue with Nix's central expression repository and end with describing the relevant update mechanisms.

2.2.1 Nix Expressions

The *Nix language* is a domain-specific, dynamically-typed and functional programming language. It is used to unambiguously define build steps[49:5]. The language is kept simple and will feel mostly intuitive to users already familiar with other functional languages. Listing 2.1 shows a simple expression that evaluates to the string `"the result is 3"`. Expressions of the Nix language are typically stored

[2]https://nix.dev

in files with the ending .nix. They can be evaluated using the Nix executable, which is written in C++ and will be called *CppNix*[37] from here on to avoid confusion.

Listing 2.1: Simple Nix Expression

```
let
  f = {a, b}: a + b;                # function definition, adds arguments a and b
  s = {x = 1; y = 2; z = 4;};       # attribute set (a. k. a. map, dictionary)
  result = f {a = s.x; b = s.y;};   # apply set attributes x and y to f
in
  "the result is ${result}"         # String interpolation
```

2.2.2 Components and Derivations

The primary building blocks of Nix are *components*, one or multiple files managed by Nix[24:2.1]. Components could for example be a program's executable files, a configuration file or a directory containing source code. They are stored in the *Nix store*, a subdirectory of the file system (usually `/nix/store`), which is designated for that purpose. To ensure multiple versions of the same program can coexist, Nix uses cryptographically generated names for components. How those are calculated differs and will be explained later in the chapter. These names of components are called *Outpaths*[24:2.1]. For instance, a the component Bash 5.2 might have the Outpath `/nix/store/5jw69mbaj5dg4l2bj58acg3gxywfszpj-bash-5.2` and will therefore be stored in this directory by CppNix. The executable(s) of a component are usually stored in the subdirectory `bin`, so that the full path to the Bash 5.2 executable would be `/nix/store/5jw69mbaj5dg4l2bj58acg3gxywfszpj-bash-5.2/bin/bash`.

An expression might evaluate to a *derivation*, a "specification for running an executable on precisely defined input files to repeatably produce output files"[49:5.4]. The dependencies of a build process, like a program's source code and possibly a compiler, must be specified as components (or derivations thereof) themselves. Such a 'recipe' can then be used to *realize* the component by executing it. CppNix will first realize the derivation's dependencies and provide them to the build process before performing the actual build. The produced files will then be stored at the component's Outpath.

Listing 2.2 shows an example of build instructions to produce a simple program The derivation only depends on Bash (line 1), which in turn depends on a multitude of other programs. In line 4 the desired Python print statement is written to target file to produce the component. The resulting program is a Python script that prints `Hello World!` on execution. Note that, because Python isn't used to create the build result, it is not a dependency of the component `hello`. Figure 2.2 shows a part of the dependency graph of the component from listing 2.2. To build `hello`, CppNix will first need to obtain Bash, which in turn depends on multiple other components, including Bison.

Listing 2.2: Simple Nix Build Instructions

```
{ bash }:  # list of dependencies
derivation {
  name = "hello";
  builder = "${bash}/bin/bash";
  args = ["-c" "echo 'print(\"Hello World!\")' > $out"];
  system = "x86_64-linux";
}
```

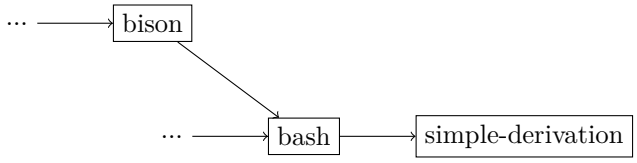


Figure 2.2: Partial dependency graph of Hello

When realizing derivations, CppNix will always first perform a lookup for the components Outpath in the Nix store. If there is a match, the derivation is not build to save work, as Nix assumes the result would not differ significantly.

2.2.3 Reproducibility

For this to work, Nix components should only depend on their defined dependencies, coming closer to the ideal of constituting a mathematical function. To help achive this, CppNix enforces constraints on the build process: either it must be performed in an isolated environment or its result must already be known before the build. Both mean, when assuming perfect isolation and deterministic computation, that Nix will always produce the same output for the same derivation (or fail).

Derivations are realized in isolated build environments by default. For non-fixed-output derivation-based components, the Outpath is defined by a calculating a hash for all input attributes, including the build script and its dependencies, and appending the component's name. The exact specification can be found in [49:10.3]. As dependencies are referred to by their Outpaths, changes in any dependency propagate to also change the components Outpath.

If however an `outputHash` is part of the derivation, CppNix will not isolate the network in the build process and instead check, that the resulting file's checksum matches the specified hash[49:5.4.1]. This is mostly used to download files, which would not be possible with full isolation in place. Build instructions with an `outputHash` are called fixed-output derivations. Contrary to non-fixed-output derivations, the Outpath of fixed-output derivations depends only on their `name` and `outputHash` attributes, but not on other attributes like their dependencies and build instructions[49:5.4.1]. Conceptually, this can be described as constituting a constant function. These properties will be relevant later to understand their limits in the context of verification.

Listing 2.3 showcases a simple fixed-output derivation. When realizing it, CppNix will download the Nix source code from the given URL using Wget. If the downloaded file does not match the provided checksum, the build fails. Figure 2.3 visualizes the derivation's dependencies, where the double border of nix-source marks it as a fixed-output derivation. As a fixed-output derivation's Outpath only depends on its `name` and `outputHash`, the Outpath of nix-source would not change if for example Curl was used for the download of the same files instead.

Listing 2.3: Simple fixed-output derivation

```
{ bash, wget }:
derivation {
  name = "nix-source";
  builder = "${bash}/bin/bash";
  args = ["-c" "${wget}/bin/wget --output-document $out https://github.com/NixOS/nix
    ↪ /archive/refs/tags/2.19.4.tar.gz"];
  system = "x86_64-linux";
  outputHash = "sha256-5pmoYP8A7XmBKbe8EN0Y+HASPphPH34U7ltbPdye0Dg=";
}
```

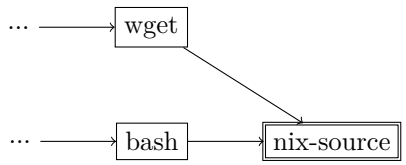


Figure 2.3: Partial dependency graph of nix-source

2.2.4 Nixpkgs and NixOS

Nixpkgs[3] is the central Nix expression collection and contains build instructions for more than 100000 packages[43, 44]. Additionally, it provides the user with Nix library functions and includes the build

[3] https://github.com/NixOS/nixpkgs

instructions of *NixOS*, a Linux distribution that can be configured through Nix expressions[27]. Nixpkgs is developed collaboratively by thousands of contributors through Github.

2.2.5 External Expressions

The Nix build system can use *external expressions*, Nix code from files that reside outside the local file system. For example, most Nix expressions import derivations from Nixpkgs to make programs available in the build process. Another common use case of external expressions is building and updating one's NixOS system from external configuration files.

The distribution of expression files can be described with a *deployment policy*[26]. While different ways of getting the source to the target can be employed, such as plain http downloads, I will focus on updates "through a version management system"[24:2.5] using the example of Git. VCSs give the user greater flexibility with respect to which version to deploy while also making the history available, which is necessary for some verification schemes described later on. In all common use cases a Git archive is provided to consumers of expression, which means this narrowing does not impact the applicability of my findings to a significant extend.

Also note, that users will usually use the transparent source/binary deployment model (substitution) described in [23], which allows them to download prebuilt binaries from a trusted source and thus speed up the build process. As this thesis is only concerned with the secure distribution of expressions, I will restrict considerations to the simpler source deployment model described in this section. Because substitution is only an optimization of the later (as per source/binary correspondence[23]), the results are valid for both models.

External expressions from Git are included into Nix code either directly though fetching functions or mediated though a locking tool. Both will be introduced in the following subsections.

2.2.5.1 Fetchers

Fetchers are Nix functions, which an user can use to reference external files. They are regularly used for including source code as well as other Nix expressions. To ensure a fetcher always evaluates to the same files, it can be locked by providing a checksum of the resulting files. This will have the fetcher always evaluate to the same files or fail, in the case they're unavailable. There are two kinds of fetchers with some similarities in usage, while differing in their implementation.

The *built-in Git fetcher* is defined in the source code of CppNix as the function `builtins.fetchGit`. Listing 2.4 demonstrates its usage. On evaluation of the shown fetcher, CppNix will download the latest Git repository from the given URL. It is then able to evaluate the outer expression, which will yield the content of the Nix Readme as a string. Built-in Git fetchers can be locked by passing a commit hash as `rev` (for revision)[49:5.6], as shown in listing 2.5.

Listing 2.4: Nix expression with built-in Git fetcher

```
{ readFile }:
readFile
  "${builtins.fetchGit {url = "https://github.com/nixos/nix.git";}}/README.md"
```

Listing 2.5: locked built-in Git fetcher

```
builtins.fetchGit {
  url = "https://github.com/nixos/nix.git";
  rev = "5b99c823ef95ba5c642ae105815d5acd4f093aa3";
}
```

The *Nixpkgs Git fetcher* looks similar to the user. However, it is implemented as an expression in Nixpkgs, instead of as part of CppNix. Because fetchers are typically supposed to make external files available, they must have Network access to download them. Consequently they typically evaluate to a fixed-output derivation and therefore require a checksum. Listing 2.6 contains a locked Nixpkgs Git fetcher. As can be seen, the function is named similarly to the built-in fetcher (`fetchGit` and

`fetchgit`). Line 5 contains the checksum that is passed to signal CppNix that network isolation is not required for this build. The whole expression will evaluate to the Readme of Nix at the given commit.

Listing 2.6: Nix expression with Nixpkgs Git fetcher

```
{ readFile, fetchgit }:
readFile "${fetchgit {
  url = "https://github.com/nixos/nix.git";
  rev = "5b99c823ef95ba5c642ae105815d5acd4f093aa3";
  hash = "sha256-ObaVDDPtnOeIE0t7m4OVk5G+OS6d9qYh+ktK67Fe/zE=";}
}/README.md"
```

Unlocked fetchers are automatically updated if their target changes when the local cache entry expires[49:5.6]. Contrary to that, locked fetchers need to be updated manually by changing the target (i. e. to a different Git commit) and, in the case of fixed-output derivations, the respective checksum.

2.2.5.2 Locking tools

For our purposes, *Nix Flakes*[49:8.5.13] can be viewed as a locking system for Nix. To include an external file, the user can specify a location, such as a reference in a Git repository, in a format called flake URI. Flake URIs include a specification of the built-in fetcher that is used to download the content they point to. Flakes can then be locked by executing the command `nix flake lock`[49:8.5.19]. In the case of Git-based URIs, this will lookup the commit that the specified reference is pointing to. It will then save the hash of the commit to a file called `flake.lock`. Nix can then continue to use the locked state, even if the reference changes, as long as the commit it pointed to is still available. The user can conveniently update the locked commit to the current state of the reference by running `nix flake update`[49:8.5.24]. Flakes are an experimental feature of CppNix[49:12.4], but widely used in the community[20].

External locking tools like Niv[4], Npins[5] and Gridlock[6] are separate programs, that allow managing external dependencies in an ergonomic way. In essence, they also provide command line interfaces for adding and updating fetchers. They behave similarly to the `nix flake` command with regard to locking, but are not interoperable with it.

2.3 Update Security

In this section, I will lay out the relevant prior work on update security. After describing some research on package manager security, I will introduce The Update Framework, which provided the main inspiration for the threat model and considered attacks of this work. I will continue by outlining Git-specific Metadata Manipulation Attacks and then Gittuf, a tool with similar aims which will be used in chapter 4 and chapter 5. The section will close with a description of security mechanisms of package managers displaying similarities to Nix.

[12] provides an analysis of ten popular package managers and their vulnerabilities against repository compromises, where a repository consists of packages and matching metadata. Cappos et al. provide a list of attacks and analyze whether they are prevented by the security mechanisms of each package manager. They derive a hierarchical classification scheme based on which part of the package distribution chain is signed by the providers, which I will later use for my analysis of Nix. In this scheme, not signing any files sets the baseline as being the least secure distribution practice. Package managers that require repositories to sign their root metadata file (a. k. a. repository index) are considered the most secure design.

In [13] the same authors provide insights into package manager architecture based on an analysis of Apt and Yum. They derive rules to follow when distributing packages and tie each viable attack on a package manager back to a violation of a rule. The authors further conceptualize three design principles for secure package managers,

[4] https://github.com/nmattia/niv
[5] https://github.com/andir/npins
[6] https://github.com/lf-/gridlock

- selective trust delegation
- customized repository views and
- explicitly treating the repository as untrusted

, which will be included in the discussion in chapter 5.

[2] also analyzed a series of package managers with a focus on the Perl Package manager Cpan and derive general recommendations from their analysis. However, these mainly boil down to ensuring correctness of the implementation and do not go into conceptualization.

Contrary to this narrow focus, the Authors of [6] encompass all ways a package manager's security may be at risk, which they categorize into three groups:

- malicious registries, where the threat originates from the package registry itself,
- malicious developers, where vulnerabilities are introduced upstream, and
- malicious integration services, where attacks are possible through code pipelines.

The article then introduces a concept to solve all three vectors with the downside of missing backwards-compatibility. This thesis will only focus on attacks from malicious registries and preventions for the preexisting Nix infrastructure.

2.3.1 The Update Framework

The Update Framework[11, 56] (TUF) establishes a concept for designing secure software update mechanisms. It builds on the work in the previously mentioned literature and aims to prevent the (total) compromise of an update client, even if the update server or a part of the trusted developers were compromised. TUF intends to secure against the compromise of some signatures by:

- distinguishing between different roles
- enabling threshold signing
- allowing selective trust delegation and
- using validity periods to detect freeze attacks.

As noted by [18], TUF was designed with binary distributions in mind, and some of the problems it solves don't occur with functional models in general. However, the framework addresses a similar threat model as this thesis and is valuable literature for its general approch of auditability through signing. The specification also provides a list of plausible attacks which will be used in chapter 3.

2.3.2 Metadata Manipulation Attacks on Git

[60] introduces *metadata manipulation attacks* on Git. These target developers of a project and can be performed by attackers that control the developer's view on a synchronizing repository, either by compromising the synchronizing server, the communication channel between the developer and the synchronizing repository or by being an insider with write access. The authors demonstrate how manipulation of references can mislead developers to merge changes unintendetly, roll back the repository state or miscoordinate their work. The following chapters will show how metadata manipulation attacks may effect Nix's update mechanisms.

2.3.3 Gittuf

Gittuf[7] is a tool to protect Git repositories from supply-chain attacks. As such, it reduces the damage an attacker can do, even in the case of an sustained compromise of the central repository, which is similar to the threat model of this work. Its design goals include the protection from Git metadata manipulation attacks on the development process. To mitigate attacks, Gittuf logs the manipulation of references to history, which is synchronized through the central repository. Because the log is signed, it cannot be changed by an unauthorized party without alarming verifiers. By having each reference change be signed by a trusted developer, Gittuf ensures that an update client cannot be presented with a state that has not been authorized by the developers. When cloning a Gittuf-enabled repository, the

[7] https://gittuf.dev

program can compare the initial root keys of the downloaded repository against preknown keys. This can be used to bootstrap trust.

Gittuf is based in academic research[60] and under active development. It supports a multitude of signature schemes and can be used to grant fine-tuned permissions to developers. I will show how Gittuf can be used to protect Nix expression updates and discuss use cases.

2.3.4 Other Package Managers

Package managers are used for installing, updating and removing software on a system[55] and are often entwined with the operation system. This subsection will showcase update security mechanisms from other source-based package managers to show how they approached the same problem.

Portage[8] is the default package manager of Gentoo[22:12]. It is a source-based package manager and provides its package descriptions (Portage tree) through a central Git repository, similar to Nix. Each commit in this repo is signed with the committing developer's PGP key[30]. To verify the authenticity of the signing keys, the user is expected to employ known techniques such as a web of trust[14]. This has the advantage of not requiring a central authority and instead allowing the user to specify whom they trust. However, it presents no structured approach to verifying historic commits[18] and is more difficult to use[5:3.10].

Pacman, the package manager of Arch Linux[22:11], also uses PGP keys, although with 5 keys considered universally trusted. Those central identities can collectively delegate trust to package maintainers, who in turn sign the released packages[47]. The package index however is not signed and therefore vulnearble to manipulation, which means Pacman should be considered insecure against attacks by malicious registries[12].

A couple of language-specific repositories moved towards partially adopting TUF in their distribution mechanism, such as Haskell's Hackage[19] and Python's PyPI[39]. For Ocaml's Opam [46] proposed a concept for how TUF could be integrated into the package manager. The authors also present with Conex a client, that is able to verify packages and assist developers in signing them. As of yet, the mechanism is not used by the official repository.

Guix[9][17], a functional package management tool heavily inspired by Nix, already offers independent cryptographic verification of updates to its central package repository. The introductory article [18] provides a tool to check the security constraints of the Guix project, which is used in the default Guix update mechanism. Essentially, the repository includes a file containing the public keys of all developers. For a commit to be deemed verified, it must be signed by at least one key from each of its parent commits. The article offers a generalization of the tool for other projects, but also mentions how different workflows might hinder a direct transfer. Chapter 9 provides some context on other build systems and frameworks, including Nix. Git-Verify, which will be described in chapter 4, is based on the commit verification of Guix.

Expression verification was also previously discussed by the Nix community[35, 40, 48, 57, 61]. However, efforts to implement a Guix-style verification mechanism for Nixpkgs were abondoned as it was deemed not to fit Nixpkgs' contribution workflow[54]. Expression verification for Nix Flakes remains an open issue[25].

[8] https://wiki.gentoo.org/wiki/Portage
[9] https://guix.gnu.org

Chapter 3

Attacks on Nix Updates

By making the path from source code to deployment more direct and auditable, Nix can offer robust security properties to the user[23]. Nonetheless, in this chapter I will point out some limitations of Nix's current security model regarding the distribution of expressions. My study assumes a user who is unable to audit changes on every expression update, but instead wants to delegate trust to the expression's developers. After general remarks on development and distribution mechanisms, I will outline the considered threat model and define the security goals. I will then list the attacks on update integrity from literature and show whether they are relevant for the stated threat model.

3.1 Attacks on Development and Distribution of Nix

Generally speaking, development and provisioning of the code can be viewed as separate steps. For example, Guix develops their central expression repository through a mailing list[1][18] and provides it to users as Git repository and tar archive[2] for consumption. Nix code is often developed using a central forge (e. g. Github) and distributed in a multitude of formats, including Github API file downloads and tar archive downloads.

When developing code using a central repository, it is sensible to use that to distribute it as well, as setting up a separate distribution mechanism would increase the attack surface. However, the central repository still remains as a single point of failure that could be attacked to compromise the software's development or its distribution. In the introductory thesis[24:6.1] the author writes about Nix's trust model:

> *We also assume that the set of expressions has been received unmodified […], meaning that they have not been modified by a "man in the middle".*

Developers typically won't provide the expression directly from their computer, but instead have a Git forge distribute it in their place. Although the code is usually fetched from the forge via a protected channel, the forge itself or a malicious actor with access to it might alter the expression.

[24] suggests using cryptographic verification to protect against manipulation. However, until now Nix did not have a structured approach to mitigate this danger as its tools did not support digital signature verification. This thesis proposes changes to do so.

3.2 Threat Model

I will consider a Nix expression that is developed cooperatively by multiple developers through a central Git repository. The users regularly update the expression from the central repository (or a fork thereof). The update is done by fetching a specific Git reference (e. g. `heads/main`) from the central repository and using the commit it points to. It might be done automatically and unsupervised.

[1] https://lists.gnu.org/archive/html/guix-commits
[2] https://git.savannah.gnu.org/cgit/guix.git

I further assume that an attacker has write access to the central Git repository (creating a malicious registry[6]). The attacker may have achieved this by

- compromising the forge server,
- performing a machine-in-the-middle attack on the communication between the forge and a developer or update client or
- being handed access to the repository.

Depending on the concrete scenario the attacker is able to present an altered view of the repository or deny access to it to some or all developers and clients. The attacker does not have access to the developer's machines or their private keys.

3.3 Security Goals

A given security system should ensure update integrity, meaning if an update is successful, the new code is the latest that was intended by the developers to be distributed. In any other case the update should fail.

The availability and confidentiality of the update as well as the availability of the clients will not be considered.

3.4 Viable Attacks

The latest TUF specification[11] lists the following attacks on update integrity (selection by the author, formatting added):

> - Arbitrary installation attacks*: an attacker cannot install anything they want on the client system. That is, an attacker cannot provide arbitrary files in response to download requests.*
> - Extraneous dependencies attacks*: an attacker cannot cause clients to download or install software dependencies that are not the intended dependencies.*
> - Indefinite freeze attacks*: an attacker cannot respond to client requests with the same, outdated metadata without the client being aware of the problem.*
> - Mix-and-match attacks*: an attacker cannot trick clients into using a combination of metadata that never existed together on the repository simultaneously.*
> - Rollback attacks*: an attacker cannot trick clients into installing software that is older than that which the client previously knew to be available.*
> - Vulnerability to key compromises*: an attacker, who is able to compromise a single key or less than a given threshold of keys, cannot compromise clients.*
> - Wrong software installation*: an attacker cannot provide a file (trusted or untrusted) that is not the one the client wanted.*

As mentioned before, some of the problems it tackles don't occur in Nix's functional and source-based model, because TUF was created for artifact-based updates.

- Extraneous dependency attacks cannot be performed, because Git commit objects are content addressable. If the correct commit is retrieved from the server, all build dependencies are resolved correctly by Nix.
- Wrong software installation attacks are a conceptual mismatch with source-based models.

Notice, that the presence of protection against an arbitrary installation attack is assumed, because if this is not given, all security mechanisms against attacks via the repository are unnecessary. When taking into account the metadata manipulation attacks from [60] this leaves the following attacks:

- arbitrary installation attacks
- indefinite freeze attacks

- mix-and-match attacks
- rollback attacks
- key compromises
- Git metadata manipulation attacks

Section 2.3 introduced a classification of package managers by their usage of signatures from [12]. By referring to this classification, Nix must be considered a "package manager without security", as it fully trusts the repository and does not have mechanisms to protect against its manipulation. This corresponds to having no protection against TUF's arbitrary installation attack. However, by implementing a signature scheme, as recommended by numerous literature [4, 12, 24, 56] and proposed by this thesis, it could easily become a package manager with "root metadata signatures", the highest security level of the classification. The remainder of this thesis will discuss how such a mechanism could be implemented to protect from the outlined threats.

Chapter 4

Prevention mechanisms

When considering the central synchronizing repository untrustworthy, as done by our threat model, the user needs to be able to verify, the content has not been modified on the path from the developer's machine to theirs. As described in chapter 3, cryptographic signatures are a well-established mechanism to do so. This chapter introduces three tools that use signatures to verify Nix expression updates. In essence, they provide the user with ways to reconstruct and verify the development process algorithmically. They differ, amongst others, in the scope of what is verified to be authored by a trusted developer:

- Nix commit verification ensures that a commit was authored by an authorized developer
- Git-Verify additionally ensures that all ancestors were authored by authorized developers
- Gittuf additionally ensures that the whole development process was protected against metadata manipulation attacks

However, signature verification cannot guard against rollback attacks, as an attacker does not attempt to modify files but merely serves an old state as the newest one. Therefore, I additionally modified the Nix locking system Niv to enable the detection of rollback attacks, which I will describe at the end of the chapter.

4.1 Nix Commit and Tag Verification

One simple way to provide better update security to Nix deployments is to integrate checking Git commit and tag signatures in Nix's Git fetchers.

As part of this work I implemented Commit signature verification into CppNix[1]. The feature is available for the built-in Git fetcher through the `verified-fetches` experimental feature since version 2.19[49:12.4]. It can be enabled by passing `verifyCommit = true` to the fetcher. Additionally, SSH keys are passed to the argument `publicKeys` as a list of attribute sets, each containing `key` with the public key and `type` specifying its type. As a convenience, if only a single key should be specified, it can be passed as `publicKey`. Listing 4.1 shows a fetcher with commit verification using a single key.

Listing 4.1: built-in fetcher using commit verification

```
builtins.fetchGit {
  url = "https://github.com/NixOS/nixpkgs.git";
  verifyCommit = true;
  publicKey = {
    key = "AAAAC3NzaC1lZDI1NTE5AAAAIGAUlj612fQ2q9wPiogNJfMWnTeffOAPdsH46fgla04I";
    type = "ssh-ed25519";
  }
};
```

[1] https://github.com/NixOS/nix/pull/8848

During fetching CppNix usually selects a commit, either the one that was explicitly specified via the `rev` argument or the one derived from a passed reference. The only case, where Nix does not select any commit when fetching Git repositories are when fetching repositories on the same file system, that incorporate uncommitted changes and `ref = HEAD` was passed. In this case, commit verification will yield an error, because there is no commit that could have been signed. Otherwise, the selected commit is examined for a Git commit signature. If no signature is present, then Nix throws an error. Otherwise, the signature is checked against `publicKeys`. If any private counterpart of the keys can be proven to have produced the signature, the fetching continues. If no key matches, Nix will throw an error and halt the build.

Git tag verification is not yet implemented in the Nix built-in Git fetcher, but could follow the same basic mechanism. It might be enabled by passing `verifyTag = true` to the built-in Git fetcher. If `rev` was specified or if `ref` does not name a reference pointed to a Git tag object, the function would throw an error. Otherwise, Nix would ensure that the tag object is signed and the signature matches a key in `publicKeys`.

Commit and tag verification could also be transferred to the Nixpkgs Git fetcher with the same parameters and semantics. Importantly, the verification would have to be perfomed outside of the fixed-output derivation for reasons discussed in section 5.1.5.

4.2 Git-Verify

Git-Verify[2] implements the Guix verification mechanism[18] and expands it with a number of features to enable usage and adoption in Nixpkgs and other large Nix projects.

In this section, I will first lay out the requirements on a repository for Git-Verify to be able to function. Next, I will provide a formalization of how the program decides what is considered trustworthy, before commenting on the implementation details and enhancements.

4.2.1 Requirements for Verification

As with the Guix verification mechanism, a Git-Verify secured repository must contain a *committers file*, holding the necessary information for verifying following commits. By default, Git-Verify assumes the file is named `committers.json` and is located at the root of the repository. The file format chosen for Git-Verify is based on JSON and looks as follows:

```
{
    "committers": {
        <committer 1 name>: {
            "email": <committer 1 email>,
            "publicKey": <public key>,
        }
        ...
    }
}
```

The committers' name and e-mail address must match their commit settings. Currently only SSH Ed25519 keys[34] are accepted as public keys. Listing 4.2 shows a minimal example of a committers file, which forbids any modification from anyone other than Alice. The committers file in listing 4.3 also allows modifications from Bob.

To confine the impact of a compromise, Git-Verify makes it possible to restrict permissions of committers. For this, any committer definition can contain an additional attribute `allowed`, that lists which files and directories the committer can change. Permissions to change a file also include file creation and deletion. If an entry ends with `/`, it is considered a directory and the commiter can change all contained files and subdirectories. If the attribute is not defined, the committer will be allowed to change any file.

[2] https://codeberg.org/flandweber/git-verify

Users may also additionally define one of `protected` and `unprotected` as a top level attribute, which also must be a list of files and directories. `protected` causes the verification mechanism to only include its file paths. `unprotected` causes them to be excluded from it respectively. If neither is given, then all files will be considred protected.

Listing 4.4 displays a committers file with a more granular access control. Additionally to the two files `committers.json` and `README.md`, only the folders `src/` and `doc/` are protected in the first place. All other files and folders may be changed without authentication without having the verification fail. Of the protected files, Bob can only change `README.md` as well as all files in `src/submodule/` and `doc/`, while Alice can modify all files.

Listing 4.2: simple committers file

```
{
  "committers": {
    "alice": {
      "email": "alice@example.com",
      "publicKey": {
        "type": "ssh-ed25519",
        "key": "AAAAC3NzaC1lZDI1NTE5AAAAIGAUlj612fQ2q9wPiogNJfMWnTeffOAPdsH46fgla04I
    ↪ "
      }
    }
  }
}
```

Listing 4.3: committers file with multiple committers

```
{
  "committers": {
    "alice": {
      "email": "alice@example.com",
      "publicKey": ...
    },
    "bob": {
      "email": "bob@example.com",
      "publicKey": ...
    }
  }
}
```

In these listings the public keys are omitted for brevity.

Listing 4.4: committers file with protected files restrictions

```
{
  "protected": [
    "committers.json",
    "README.md",
    "src/",
    "doc/"
  ],
  "committers": {
    "alice": {
      "email": "alice@example.com",
      "publicKey": ...
    },
    "bob": {
      "email": "bob@example.com",
      "allowed": [
        "README.md",
        "src/submodule/",
        "doc/"
      ],
      "publicKey": ...
    }
  }
}
```

Additionally to a committers file, every verification process requires an *introduction*. An introduction is a commit that is given to the verifying client out-of-band as a trustable state and does not change. The file tree of the introduction needs to contain a committers file, because it is used to bootstrap trust in

all following commits. For the same reason the introduction needs to be an ancestor of any commit that the client is trying to verify.

4.2.2 Verification Invariant

The verification invariant describes which commits can be trusted by an updating client according to Git-Verify. It is a recursive definition that introduces trust through the introduction and can be summarized as

1. The introduction is verified.
2. All children of only verified commits are verified, if they were signed with a valid signature. A valid signature is one that was generated using a key, that is allowed to change all files in the difference between each parent and the child.
3. Any auto-merged commit with only verified parents is verified.

Formally, the set of verified commits (with respect to introduction i) $Verified_i \subseteq Commits$ can be defined as in eq. (4.1),

$$i \in Verified_i \tag{4.1a}$$

$$\forall c \in Commits : (\forall p \in Parents_c : p \in Verified_i \land valid(p, c)) \Rightarrow c \in Verified_i \tag{4.1b}$$

$$\begin{aligned} \forall c \in Commits : (&Parents_c \subseteq Verified_i \land |Parents_c| \geq 2 \\ &\land automerge(Parents_c) = Tree_c) \Rightarrow c \in Verified_i \end{aligned} \tag{4.1c}$$

where $valid : Commit \times Commit \to \mathbb{B}$ describes whether a commit was made according to protocol. To define $valid(p, c)$, let $changed = \{file | (file, hash) \in Tree_p \triangle Tree_c\}$, the set of files that were changed from p to c. Further let $changedProtected$ describe the set of all protected files in $changed$, according to the committers file in $Tree_p$. Then $valid(p, c)$ is $True$ iff

- $changedProtected = \emptyset$ or
- c is signed by a public key k and the committers file in $Tree_p$ gives the owner of k permission to change all files in $changedProtected$.

4.2.2.1 Example 1

Figure 4.1 shows a simple example of a commit graph with committers files. Each commit's committers file is referenced in the brackets. The names on the edges describe who committed and signed the commit. Protected files and restrictions of committers are not part of this example.

We shall now determine the set of verified commits according to the verification invariant as stated above. For this, $c1$ will serve as the introduction.

- $c1 \in Verified_{c1}$, because the introduction is always verified (eq. (4.1a)).
- $c2 \in Verified_{c1}$, because it was created and signed by Alice and they are in $c1$'s committers file.
- $c3 \notin Verified_{c1}$, because it was created by Bob, who is not part of $c1$'s committers file.
- $c4 \in Verified_{c1}$, because Alice changed the committers file with $c2$ to include Bob, allowing him to commit $c4$.
- $c5 \notin Verified_{c1}$, because its parent $c3 \notin Verified_{c1}$.

Therefore $Verified_{c1} = \{c1, c2, c4\}$.

4.2.2.2 Example 2

Figure 4.2 introduces a slightly more complex example, that includes fine-grained control through protected files and restrictions. Table 4.1 lists which files were changed by each commit. To reiterate, listing 4.2 only allows changes from Alice while listing 4.4 allows changes from Bob to `src/submodule/`, `doc/` and `README.md` and anyone to folders outside of `src/` and `doc/`. As before, I will explain for each commit why it is or is not verifiable. For the verification we will choose to use $c2$ as the introduction, since it is the first commit with a committers file.

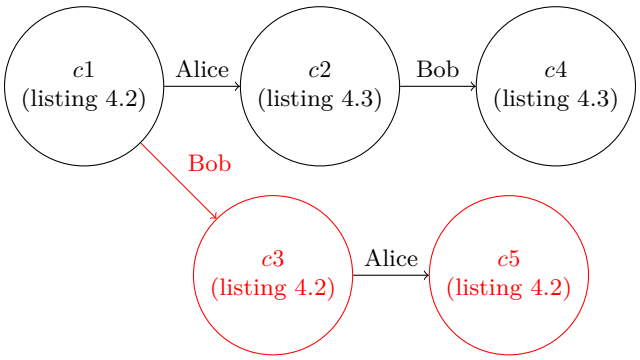


Figure 4.1: simple commit graph with committers file

- $c1 \notin Verified_{c2}$, because the introduction is not an ancestor of it.
- $c2 \in Verified_{c2}$, because the introduction is verified by definition.
- $c3 \in Verified_{c2}$, because its parent is verified and its committers file allows Alice to change any files.
- $c4 \in Verified_{c2}$, because the parents committers file allows Bob to change `src/submodule`.
- $c5 \in Verified_{c2}$ for the same reasons as $c3$.
- $c6 \in Verified_{c2}$ even though it was not signed, because it does not change files and is therefore an auto-merge, which is verified if its parents are.

Therefore is in this example $Verified_{c2} = \{c2, c3, c4, c5, c6\}$.

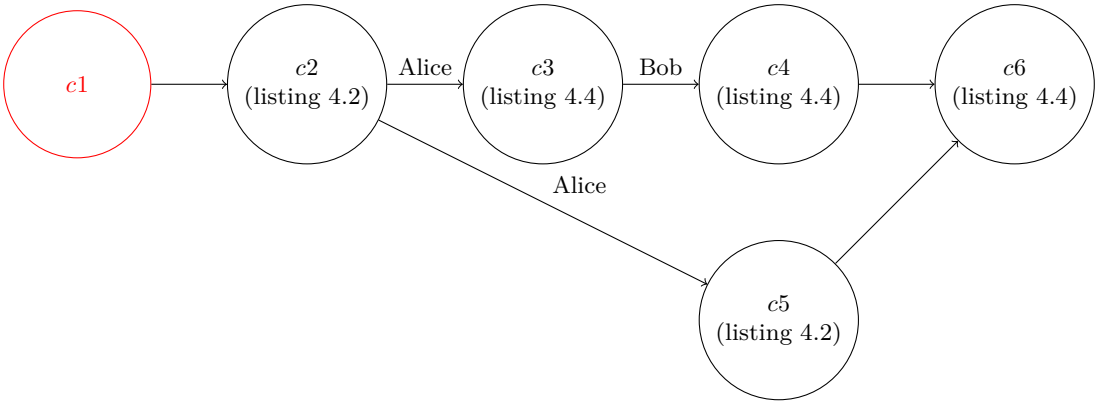


Figure 4.2: commit graph with committers file with protected files and restrictions

commit	changed files
$c1$	`src/code.nix`
$c2$	`committers.json`
$c3$	`committers.json`, `src/code.nix`
$c4$	`src/submodule/code.nix`
$c5$	`src/feature.nix`
$c6$	

Table 4.1: changed files by commit

4.2.3 Implementation

The program was implemented in Haskell[3], a general purpose, typed, functional language[45]. Due to its relative proximity to formal methods, Haskell often allows users to express concepts from theory

[3]https://www.haskell.org

with little change[59]. In this case, it allowed stating the verification invariant in an almost unmodified form, which narrowed the gap between its conceptualization and the running code. This is desirable to avoid differences between the two and therefore implementation bugs.

As described before, Git allows signing commits with multiple tools. I chose to prioritize SSH signatures in Git-Verify, due to its widespread adoption as a signature format and because multiple community members expressed their concern about the complexity of GPG[4] usage[54].

4.2.4 Git-Verify Fetcher

To make Git-Verify verification practical in some Nix deployment scenarios, I developed a Nix fetcher that is similar to the Git fetcher from Nixpkgs and automatically performs the verification.

Listing 4.5 shows how a call of the fetcher can look like. As with the Nixpkgs fetcher, the user must specify the url and version of the repository to fetch and a checksum to ensure no content was modified. Additionally the argument `intro` is passed for the verification, which must be the hash of the introduction. If the verification fails, the fetcher exits with an error.

Listing 4.5: Nix expression with Git-Verify fetcher

```
{ fetchgitverify }:
fetchgitverify {
  url = "https://github.com/nixos/nixpkgs.git";
  rev = "057f9aecfb71c4437d2b27d3323df7f93c010b7e";
  hash = "sha256-MxCVrXY6v4QmfTwIysjjaX0XUhqBbxTWWB4HXtDYsdk=";
  intro = "4ecab3273592f27479a583fb6d975d4aba3486fe";
}
```

As mentioned, it is common to use locking tools to update Nix fetchers. In order to make updates to `fetchgitverify` more ergonomic, I added it to a fork of the locking system Niv[5]. When using the forked version of Niv, update sources can easily be added by executing `niv add gitverify <repository url> --intro <intro SHA1 hash>` in a Niv-enabled project.

4.2.5 Possible Enhancements

There are multiple features that could further expand the usability and security properties of Git-Verify, but which were as of yet not implemented. In the following, I will list concepts that could be implemented in future work.

- *Sub-projects* would allow delegating permission management from a central committers file of the repository to smaller units. This would be possible by allowing an additional top-level attribute to the committers file, which specifies folders that are to be considered sub-projects. These folders would be required to contain a committers file. Changes to them would only be verified using the sub-projects committers file, which would allow them to autonomously manage their members and permissions.
- *Tag verification* would allow checking the signature of Git tags. This would help prevent some redirection attacks by giving the developer the option to sign a Git tag and having Git-Verify check the signature against a key specified in the committers file.
- *Timed tag verification* could additionally be used to prevent indefinite freeze attacks and some Git metadata manipulation attacks. By allowing developers to specify branches for which a tag object is created at a certain interval, This tag would probably be automatically created using an online key and could be similar to TUF's timestamp role.
- Because auto-merges can go unsigned, this allows any privileged attacker to conduct a merge as long as it does not produce a conflict, which might be undesirable for the user's threat model. For this reason users may want to disable the verification property that accepts any auto-merged

[4] https://www.gnupg.org
[5] https://github.com/flandweber/niv/tree/gitverify

commit with verified parents (eq. (4.1c)) through a committers file attribute. For forward-compatibility, Git-Verify already allows committer files with an optional top-level attribute with the name `automerge` and a boolean value.
- To further speed up the verification mechanism, commits, that were already verified with respect to a given introduction, could be cached locally.

4.3 Gittuf Nix Fetcher

To make Gittuf usable in the Nix context, I developed a fetcher[6] that works similarly to the Nixpkgs Git fetcher, but includes Gittuf's verification. Because Gittuf's threat model does not only protect commits, but additionally protects against reference manipulations, a verification always needs to be perfomed with regard to a specific reference. The fetcher additionally requires the projects root keys to bootstrap trust. An example use of the Gittuf fetcher can be seen in listing 4.6.

Listing 4.6: Nix expression with Git-Verify fetcher

```
{ fetchgittuf }:
fetchgittuf {
  url = "https://git.example.com/project.git";
  rev = "057f9aecfb71c4437d2b27d3323df7f93c010b7e";
  ref = "main";
  root-keys = [./project-root-key.pem]
  outputHash = "sha256-MxCVrXY6v4QmfTwIysjjaX0XUhqBbxTWWB4HXtDYsdk=";
}
```

The verification is done by calling Gittuf and consists of three steps:

1. clone the repository and verify that the root keys match
2. verify the given reference
3. verify that the given revision is part of the references history

This process saveguards against most possible attacks on update integrity as will be shown in the following chapter.

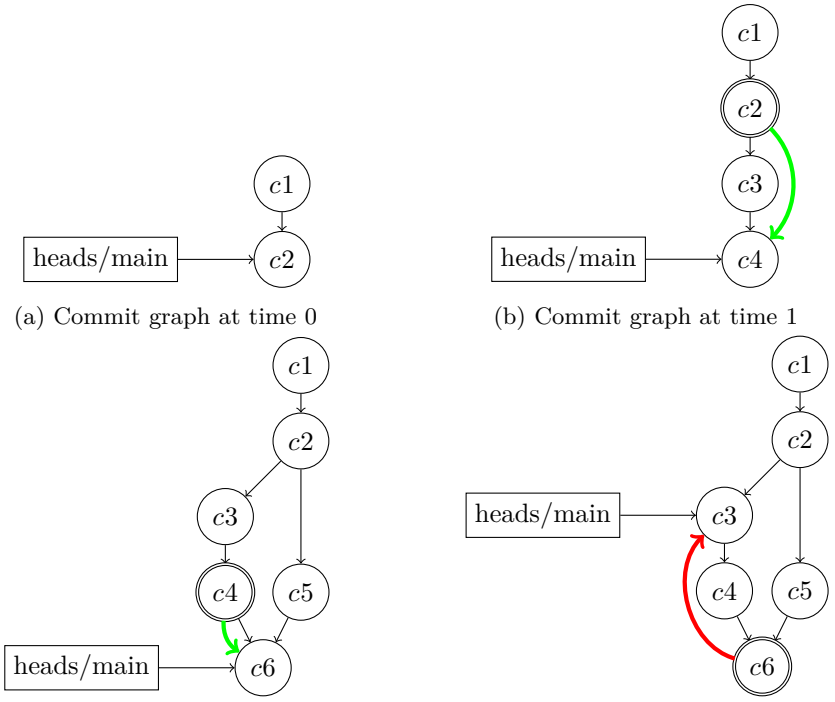
4.4 Niv Rollback Protection

The simplest way to implement rollback protection in the Git context is to ensure that the previous locked commit is an ancestor of the newly locked commit. However, Nix does not allow fetchers to access previous iterations of their package, as their behavior is not supposed to change depending on that. As this prevents rollback protection to be implemented as part of the Nix expression, it must be part of the updating program, e. g. Nix Flakes or Niv, which does have access to the previously locked state.

I implemented the additional command line flag `--rollback-protection` to the update mechanism of the third-party tool Niv[7]. It causes Niv to download the repository to a temporary folder and verifies that no rollback to a previous state occured by calling a Git command. If the verification fails, no update to the lockfile is performed.

Figure 4.3 shows an example commit graph that is modified over time. Niv tracks the main branch and tries to update to the commit it points to at every time. The doubled border marks the previously locked commit. In fig. 4.3a the reference is fetched for the first time, therefore there is no previously locked commit. $c2$ is recorded in the lockfile. In fig. 4.3b the reference moved from $c2$ to $c4$. As $c2$ is an ancestor of $c4$, the update will succeed. The same goes for fig. 4.3c, where $c4$ is updated to the merge commit $c6$. In fig. 4.3d however, the main branch is rolled back to commit $c3$. Updating $c6$ to $c3$ would violate the protection as the later is not an ancestor of the former. Therefore this update will fail if performed with rollback protection enabled.

[6] https://codeberg.org/flandweber/fetchgittuf
[7] https://github.com/nmattia/niv/pull/405



(a) Commit graph at time 0

(b) Commit graph at time 1

(c) Commit graph at time 2

(d) Commit graph at time 3

Figure 4.3: Commit graphs of update with rollback protection

Chapter 5

Discussion

After describing possible attacks in chapter 3 and introducing mitigations in chapter 4, this chapter will cover the merits and limits of the suggested security tools. The first section will dive into the verification solutions and their Nix interfaces, whereas section 5.2 will discuss rollback protection in the Nix update process.

5.1 Comparison of Verification Tools

In the following, I will list the benefits and drawbacks of each of the three considered verification tools for securing Nix updates: Nix commit verification, Git-Verify and Gittuf. First I will take a look at their security guarantees. I will go on to assess the solutions according to additional criteria, which I derived from my review of the discourse[40, 48, 54, 61] as well as conversations with individuals from the Nix community.

5.1.1 Security Benefits

Table 5.1 gives an overview of the security benefits of the proposed fetchers. All solutions protect against arbitrary installation attacks and thus reduce the ways an attacker can introduce malicious code to the client's system. Projects that use Git-Verify for commit verification are vulnerable to some mix-and-match attacks through the auto-merge verification as described in section 4.2.5. As can be seen, Gittuf is the only mechanism that can fully protect against developer key compromises by employing threshold signing. Git-Verify still holds some protections in the case of a developer compromise, because it allows owners to administer fined-grained access control. Gittuf is also the only mechanism to prevent metadata manipulation attacks on the repository's development and distribution as the other solutions only consider the commit graph.

Attack	fetchGit with commit verification	fetchgitverify	fetchgittuf
arbitrary installation attack	✓	✓	✓
indefinite freeze attack	✗	✗	✗
mix-and-match attacks	✓	✗[1]	✓
rollback attack	✗	✗	✗
key compromise	✗	✓[2]	✓
Git metadata manipulation attack	✗	✗	✓

[1] Some signed states could be merged through auto-merges.
[2] Git-Verify only protects parts of the repository in which the compromised developer has no permissions in.

Table 5.1: security benefits of Nix commit verification, Git-Verify and Gittuf

As described protection against rollbacks and indefinite freeze attacks cannot be implemented by any fetcher on its own, but will have to be enabled through the executing environment, i. e. the Nix Flakes command or a third-party locking tool.

All three fetchers can help Nix honor the design principles laid out in [13]. The principal of selective trust delegation is implemented by Git-Verify and Gittuf, while it cannot be performed using Nix commit verification. Customized repository views are partially already possible through Nixpkgs' concept of overlays[50], but are not considered as part of this work. The recommendation to explicitly treating the repository as untrusted however is central part of it and all three techniques further Nix's adherence to it significantly.

5.1.2 Nixpkgs Contribution Workflow Support

With a significant backlog of pull requests[1] it is of great importance that any solution adds as little work for the contributors and maintainers as possible. Ideally, the tool should preserve the Github web GUI based contribution workflow of Nixpkgs to the widest extent possible. I will now first describe the relevant parts of a typical contribution workflow and then mention changes, that the three tools would require. For simplicity, a branch `b` in a given repository `r` of user `u` is denoted as `u/r/b`.

A typical contribution to Nixpkgs consists of the following steps

1. A forks `NixOS/Nixpkgs` to `A/Nixpkgs`
2. A pushes new commits to `A/Nixpkgs/fix-branch`
3. A creates a Github pull request (pr) from `A/Nixpkgs/fix-branch` to `NixOS/Nixpkgs/master`
4. A adds tag `backport release` to indicate that the change should be made to branch `release` as well
5. B reviews the pr and requests changes in a comment at the Github web GUI
6. A makes changes to `A/Nixpkgs/fix-branch`
7. B reviews the pr again and approves it
8. B merges the commits to `NixOS/Nixpkgs/master` in one of three ways via Github's merge queue:
   a. Merge commit
   b. Rebase and fast-forward merge
   c. Sqash merge commit
9. Github's CI automatically opens a Github pull request from `NixOS/Nixpkgs/backport-<pr number>-to-release` to `NixOS/Nixpkgs/release`
10. B merges the commits in one of the three ways via Github's merge queue

All solutions except Git-Verify require all commits touching protected files to be signed, which means developers could not continue to merge through the Github interface as in step 8, if the commit affects protected code. Git-Verify, by supporting auto-merges, allows merging via merge commit (8a) through the Github web GUI, but also cannot support the other two merge-styles.

All solutions additionally require B (the maintainer) to sign and recommit the changes and push them to `A/nixpkgs/fix-branch`, if A is not allowed to make these changes, which is an additional step in the review process and could slow down contributions to Nixpkgs. B also needs to repeat the offline signing for backports (10).

5.1.3 Nix Update Flow Support

As users require or prefer specific ways, the tooling should support a broad range of ways to include external expressions. As explained in section 2.2.5, Nix expressions that are fetched via Git are usually updated in of the following ways:

1. Flake update
2. Built-in fetcher update as
   a. refresh of remote reference (for unlocked fetchers) or
   b. argument update (for locked fetchers)
3. Nixpkgs fetcher argument update

[1] https://github.com/NixOS/nixpkgs/pulls

4. Third-party locking tool update

Nix commit verification cleanly integrates into Nix Flake updates and the built-in Git fetcher. The other methods cannot support those two, because of the limitations on what is exposed to custom fetchers by Nix. Support for Git-Verify verification could be added by optionally having the built-in Git fetcher preserve a deterministic part of the Git history of the fetched repository. Nix built-in support for any verification method could be enabled by making the fetch process more generic, e. g. though user-defined hooks.

All verification workflows could be supported by Nixpkgs fetchers. I showed this for Gittuf and Git-Verify by implementing the according functions and described, how support for basic commit verification could be added to the existing Nixpkgs Git fetcher. The new fetchers could be supported by third-party locking tools to make updates more ergonomic, as shown at the example of `fetchgitverify` and Niv.

5.1.4 Key Management

For bigger projects such as Nixpkgs, it is important to have a way of adding new developers to the project and invalidating keys. A process to do so is integrated into Git-Verify and Gittuf, while Nix commit verification lacks it. Therefore only Git-Verify and Gittuf may be suitable for providing update security to expression collections with requirements for central key management.

5.1.5 Verification in fixed-output derivations

Regardless of their capabilities there are two distinct ways of implementing verifying fetchers in Nix. The difference lies in how the separate steps of downloading and verifying relate. For the explaination, I will distinguish a Git fetchers *payload*, which will be the files referenced by a commits tree, from the *verification data*, which are the additional files required for verification. For all aforementioned tools, the verification data is a subset of the files in the .git directory of a given repository. As almost all fetchers depend upon network access at build time for the file download, they typically contain a fixed-output derivation and thus require a hash. In essence, the difference between the ways of implementing verifying fetchers is, whether the hash is only calculated on the payload or if it also includes the verification data.

The naive way to write a verifying fetcher is to have downloading and verification be part of the same derivation (without network isolation). Because the output of the derivation will typically only be the payload, it is trivial to have the fetcher always build the same files and thus match the checksum. However, recall that a derivation is never realized, if their Outpath already exists in the Nix store. Additionally, remember that fixed-output derivations depend only on the attributes `outputHash` and `name` for the calculation of their Outpath. Therefore a build process will not be run, if a derivation with the same `outputHash` and `name` is already in the store, even if it was acquired through different build instructions. This is the case because Nix does not have a concept of verification and thus doesn't care *how* files were build. The substitution of builds through preexisting files obtained in a different way makes it impractical to implement checks as fixed-output derivations.

To mitigate this, one could (ab-)use the `name` attribute to integrate a second checksum into the Outpath of a fixed-output derivation. The second checksum could not only depend on the derivations `outputHash` and `name`, but also on its build instructions and dependencies and therefore simulate a non-fixed-output derivation. This would protect against accidental substitutions through derivations with instructions an dependencies, as shown by [33]. Nevertheless, it is rather a hack that is not supported by Nix's model. This has the effect, that it cannot protect against scenarious where an attacker causes Nix to build a non-verifying fetcher to prevent verification through a verifying fixed-output fetcher on purpose.

Therefore, the download of files and verification should instead be split up into two components whereever possible. In this pattern, payload and verification data are downloaded through a fixed-output derivation. Then, checks on the files are run in a separate, isolated build process, which copies the payload to its output if they succeed. As the verification is not defined by a fixed-output derivation, CppNix will not substitute it through files definied by a different derivation. However, it requires the downloading fetcher to build a deterministic output of not only the payload, but also the verification data. Figure 5.1 visualizes the difference between the concepts as a dependency graph.

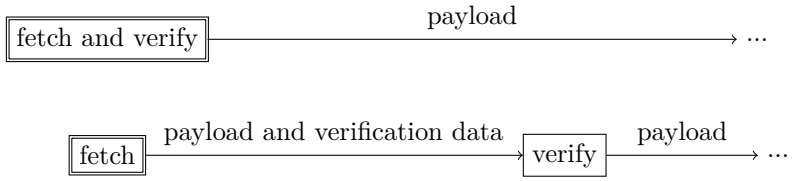


Figure 5.1: Dependency graphs of fixed-output verifying fetcher and proper verifying fetcher

The Nixpkgs Git fetcher does have attributes to download a commit including its history deterministically[50]. As Git-Verify only depends on those as verification data, I was able to let the Nixpkgs Git fetcher handle the download of the required files and implement the verification of `fetchgitverify` based on those. This can be seen in its dependency graph in fig. 5.2, where fetchgit is a fixed-output derivation and fetchgitverify is a non-fixed-output derivation that depends on it.

Gittuf however requires additional files from the .git directory as verification data, that cannot be deterministically downloaded by the Nixpkgs Git fetcher. I therefore implemented the check in `fetchgittuf` as part of the fetchers fixed-output derivation, as shown in its dependency graph in fig. 5.3. This makes the fetcher vulnerable to the described substitution problem and it is therefore not secure to rely on it for verification yet. The fetcher could be improved by integrating a second checksum as part of its `name` like described in [33]. It could be fixed properly by developing a method to download the files required for verification deterministically and seperate this into its own component.

Fundamentally it can be asserted that verifications should never be performed as part of fixed-output derivations. This may complicate implementations of verification processes significantly, but also furters reproducibility as verifications cannot implicitly depend on online recources (like key revocations).

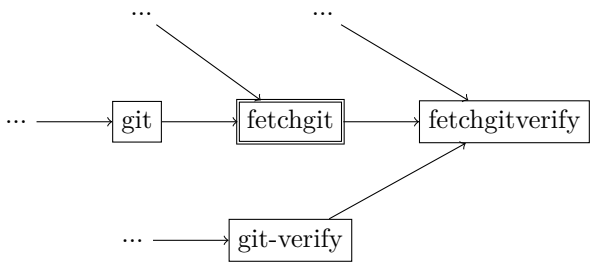


Figure 5.2: Partial dependency graph of fetchgitverify

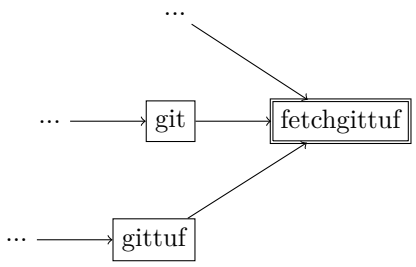


Figure 5.3: Partial dependency graph of fetchgittuf

5.2 Rollback Protection

As signature verification makes it more difficult for an attacker to create new files that are accepted by clients, they may instead try to serve an old, vulnerable version of a signed file. To prevent this, I suggested an implementation of rollback protection in the third-party locking system Niv in section 4.4. In the following I will first discuss the addition's characteristics as Git-based and then examine it as a stateful protection mechanism in Nix.

5.2.1 Git-based Rollback Protection

As Git as a VCS already provides versioning of source code, it is natural to use these for determining whether a rollback took place. This does, however, enforce that the tracked reference is treated as append-only by update clients. If a developer removes commits from a reference that is tracked by a client (e. g. the main branch of a public repository), they risk that a client had already updated to it an will therefore be unable to download the latest state. Yet, as removing commits from collaborative branches is usually discouraged[15:3.6], this matches how many projects manage their commit history.

Additionally, clients may want to change the location they update from, for example to a different branch or a fork of the reporitory. The current implementation of rollback protection in Niv requires manual intervention, if the last updated commit is not an ancestor of the commit the new location points to. This behaviour might be unexpected by the user.

5.2.2 Satefulness

As discussed before, stateful protection mechanisms cannot be implemented as Nix expressions, but instead need to be part of the updating program. Becaus the guard against rollback attacks shown in this thesis is inherintly stateful, this restriction applies to it. Such a mechanism can be effective, but has several disadvantages over a solution implemented in the Nix language or CppNix.

An obvios disadvantage is, that it requires implementing rollback protection in every update mechanism, while solutions implemented in Nix are more general and don't depend directly on whether the user uses Flakes, another locking system or updates fetchers manually. But even if they are implemented in the relevant update mechanism, the check is only run by the updating party. Contrary to the signature verifications above, it is not run at every rebuild. To understand the problem, consider a scenario where a user consumes an expression that depends on other inputs. When the developer runs an update of the expression with rollback protection enabled, the user must trust that the developers update mechanism ran the check. Of course they could manually check if the update really does not constitute a rollback, but it is not obvious from the diff of the expression files.

To move the verification process from the update tool to the fetchers, one could instead implement rollback protection by adding a `ancestors` attribute to the Git fetchers. The attribute would take a list of commit hashes and cause the fetcher to check, if all of them are part of the ancestors. An update with rollback protection would then take the formally specified commit hash and add it to the list passed to `ancestors`. With this, the user could easily see in any diff, if an update was made with rollback protection enabled or not. Additionally to the added transparency, this implementation would speed up the updates process, because the locking tool would merely move the commit hash and not download the repository to verify the property itself.

Chapter 6

Conclusion

In this work I demonstrated that Nix is vulnerable to several well-known attacks as it has no mechanism to verify whether given update comes from a trusted developer. I introduced three ways to use signature verification to mitigate attacks and discuss their security and usability.

Nix commit verification provides basic protections from repository-side attacks. It does not have key managements built-in and therefore can only be used for projects with few developers. Of all solutions Nix commit verification is best integrated in different update flows.

Git-Verify uses Git commit signatures as well, but also provides key management and granular protections. It was built to support GUI-based merging and constitutes a simple solution for moderately-sized and large projects, such as Nixpkgs.

Gittuf can principally defend against most attack vectors. It can employ a multitude of signature schemes and provides large projects with the ability to fine-tune permissions. However, because the implemented Nix fetcher verifies the expression as part of a fixed-output derivation, a user should not rely on it.

All proposed tools decrease the abilities of unauthorized attackers, albeit to varying degrees. Nevertheless, build-time verification cannot protect against freeze and rollback attacks, as protections against them must be implemented statefully. I showcased how rollback protection can be provided as part of update tools using the example of Niv. Yet in some scenarios it might be undesirable for users to rely on the update tool for protection. Freeze attacks remain a threat.

In general, it is noteworthy that some security mechanisms, such as stateful protection and key revocation, conflict with Nix's functional design. These learnings draw into question whether an expansion of the Nix's design is warranted in order to cleanly integrate them into the respective workflows. As efforts to harden supply chains intensify, expression update security will become more relevant to the Nix ecosystem. This thesis demonstrates that Git can provide a good basis for adding verification to Nix updates. Future work could refine the suggested tools, propose formal verification of the mechanisms and implement protections against indefinite freeze attacks as well as consider update availability.

References

[1] Johan Abildskov. 2020. *Practical git: Confident git through practice*. Apress L. P., Berkeley, CA.

[2] Anish Athalye, Rumen Hristov, Tran Nguyen, and Qui Nguyen. 2014. *Package manager security*.

[3] Justin Bedő, Leon Di Stefano, and Anthony T. Papenfuss. 2020. Unifying package managers, workflow engines, and containers: Computational reproducibility with BioNix. *GigaScience* 9, 11 (November 2020). https://doi.org/10.1093/gigascience/giaa121

[4] Anthony Bellissimo, John Burgess, and Kevin Fu. 2006. Secure software updates: Disappointments and new challenges. In *First USENIX workshop on hot topics in security (HotSec 06)*, July 2006. USENIX Association, Vancouver, B.C. Canada. Retrieved from https://www.usenix.org/conference/hotsec-06/secure-software-updates-disappointments-and-new-challenges

[5] Bellovin, Schiller, and Kaufman. 2003. *Security mechanisms for the internet*. RFC Editor; Internet Requests for Comments; RFC Editor. Retrieved from https://www.rfc-editor.org/rfc/rfc3631.txt

[6] Fraser Brown, Ariana Mirian, Atyansh Jaiswal, A Notzli, and Deian Stefan. 2017. SPAM: A secure package manager. *Proc. of the USENIX HotSec* (2017).

[7] Sander van der Burg and Eelco Dolstra. 2010. Automating system tests using declarative virtual machines. In *2010 IEEE 21st international symposium on software reliability engineering*, November 2010. IEEE. https://doi.org/10.1109/issre.2010.34

[8] Chris Burr, Marco Clemencic, and Ben Couturier. 2019. Software packaging and distribution for LHCb using nix. *EPJ Web of Conferences* 214, (2019), 05005. https://doi.org/10.1051/epjconf/201921405005

[9] Bruno Bzeznik, Oliver Henriot, Valentin Reis, Olivier Richard, and Laure Tavard. 2017. Nix as HPC package management system. In *Proceedings of the fourth international workshop on HPC user support tools* (*HUST'17*), 2017. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3152493.3152556

[10] Callas, Donnerhacke, Finney, Shaw, and Thayer. 2007. *OpenPGP message format*. RFC Editor; Internet Requests for Comments; RFC Editor. Retrieved from https://www.rfc-editor.org/rfc/rfc4880.txt

[11] Justin Cappos, Trishank Karthik Kuppusamy, Joshua Lock, Marina Moore, and Lukas Pühringer. 2023. The update framework 1.0.33. *GitHub*. Retrieved November 8, 2023 from https://theupdateframework.github.io/specification/v1.0.33/index.html

[12] Justin Cappos, Justin Samuel, Scott Baker, and John H. Hartman. 2008. A look in the mirror. In *Proceedings of the 15th ACM conference on computer and communications security*, October 2008. ACM. https://doi.org/10.1145/1455770.1455841

[13] Justin Cappos, Justin Samuel, Scott Baker, and John Hartman. 2008. Package management security. (January 2008). Retrieved from https://theupdateframework.io/papers/package-management-security-tr08-02.pdf?raw=true

[14] Germano Caronni. 2000. Walking the web of trust. In *Proceedings IEEE 9th international workshops on enabling technologies: Infrastructure for collaborative enterprises (WET ICE 2000)*, 2000. 153–158. https://doi.org/10.1109/ENABL.2000.883720

[15] Scott Chacon. 2014. *Pro git* (2nd ed. ed.). Apress L. P., Berkeley, CA.

[16] Cooper, Santesson, Farrell, Boeyen, Housley, and Polk. 2008. *Internet x.509 public key infrastructure certificate and certificate revocation list (CRL) profile*. RFC Editor; Internet Requests for Comments; RFC Editor. Retrieved from https://www.rfc-editor.org/rfc/rfc5280.txt

[17] Ludovic Courtès. 2013. Functional package management with guix. *CoRR* abs/1305.4584, (2013). Retrieved from http://arxiv.org/abs/1305.4584

[18] Ludovic Courtès. 2022. Building a secure software supply chain with GNU guix. *The Art, Science, and Engineering of Programming* 7, 1 (June 2022), 1. https://doi.org/10.22152/programming-journal.org/2023/7/1

[19] Duncan Coutts. 2015. Improving Hackage security. Retrieved July 19, 2024 from https://www.well-typed.com/blog/2015/04/improving-hackage-security/

[20] Guillaume Desforges. 2023. Nix community survey 2023 results. *NixOS Discourse*. Retrieved November 4, 2023 from https://discourse.nixos.org/t/nix-community-survey-2023-results/33124/1

[21] Adrien Devresse, Fabien Delalondre, and Felix Schürmann. 2015. Nix based fully automated workflows and ecosystem to guarantee scientific result reproducibility across software environments and systems. In *Proceedings of the 3rd international workshop on software engineering for high performance computing in computational science and engineering* (*SE-HPCCSE '15*), 2015. Association for Computing Machinery, New York, NY, USA, 25–31. https://doi.org/10.1145/2830168.2830172

[22] Jose Dieguez Castro. 2016. *Introducing linux distros*. Apress. https://doi.org/10.1007/978-1-4842-1392-6

[23] Eelco Dolstra. 2005. Secure sharing between untrusted users in a transparent source/binary deployment model. In *Proceedings of the 20th IEEE/ACM international conference on automated software engineering* (*ASE05*), November 2005. ACM. https://doi.org/10.1145/1101908.1101933

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

[25] Eelco Dolstra. 2019. Flake authentication. *GitHub*. Retrieved November 4, 2023 from https://github.com/NixOS/nix/issues/2849

[26] Eelco Dolstra, Merijn De Jonge, and Eelco Visser. 2004. Nix: A safe and policy-free system for software deployment. In *LISA*, 2004. 79–92. Retrieved from https://www.usenix.org/legacy/event/lisa04/tech/full_papers/dolstra/dolstra_html/

[27] Eelco Dolstra and Andres Löh. 2008. NixOS: A purely functional Linux distribution. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming* (*ICFP '08*), September 2008. Association for Computing Machinery, New York, NY, USA, 367–378. https://doi.org/10.1145/1411204.1411255

[28] Valentin Gagarin. 2023. Nixpkgs supply chain security project. *NixOS Discourse*. Retrieved November 4, 2023 from https://discourse.nixos.org/t/nixpkgs-supply-chain-security-project/34345

[29] Gentoo Project. 2018. Infrastructure/incident reports/2018-06-28 github. Retrieved May 7, 2024 from https://wiki.gentoo.org/wiki/Project:Infrastructure/Incident_reports/2018-06-28_Github

[30] Gentoo Project. 2023. Portage security. Retrieved November 4, 2023 from https://wiki.gentoo.org/wiki/Portage_Security

[31] Git Project. 2023. Hash function transition. Retrieved November 4, 2023 from https://git-scm.com/docs/hash-function-transition

[32] Git Project. 2023. Gitformat-signature. Retrieved April 29, 2024 from https://git-scm.com/docs/gitformat-signature

[33] Dennis Gosnell. 2023. FODONUTs: Fixed-Output Derivations for Operating Network-Utilizing Tests. Retrieved July 22, 2024 from https://functor.tokyo/blog/2023-07-24-fodonuts

[34] Harris and Velvindron. 2020. *Ed25519 and Ed448 public key algorithms for the secure shell (SSH) protocol*. RFC Editor; Internet Requests for Comments; RFC Editor. Retrieved from https://www.rfc-editor.org/rfc/rfc8709.txt

[35] Jaka Hudoklin. 2016. Review security of nixpkgs commit process. *GitHub*. Retrieved November 4, 2023 from https://github.com/NixOS/nixpkgs/issues/20836

[36] Joseph R. Biden Jr. 2021. Executive order on improving the nation's cybersecurity. Retrieved from https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity

[37] KA. Retrieved July 8, 2024 from https://save-nix-together.org

[38] Markus Kowalewski and Phillip Seeber. 2022. Sustainable packaging of quantum chemistry software with the nix package manager. *International Journal of Quantum Chemistry* 122, 9 (January 2022). https://doi.org/10.1002/qua.26872

[39] Trishank Karthik Kuppusamy, Vladimir Diaz, Marina Moore, Lukas Puehringer, Joshua Lock, Lois Anne DeLong, and Justin Cappos. 2013. PEP 458 – Secure PyPI downloads with signed repository metadata. *Python Enhancement Proposals (PEPs)*. Retrieved July 19, 2024 from https://peps.python.org/pep-0458/

[40] Peter Lacey-Bordeaux. 2020. Any interest in checkings signatures while building packages? *NixOS Discourse*. Retrieved November 4, 2023 from https://discourse.nixos.org/t/any-interest-in-checkings-signatures-while-building-packages

[41] Linux Foundation. 2011. The cracking of kernel.org. Retrieved May 7, 2024 from https://www.linuxfoundation.org/blog/blog/the-cracking-of-kernel-org

[42] Julien Malka. 2024. Increasing trust in the open source supply chain with reproducible builds and functional package management. In *Proceedings of the 2024 IEEE/ACM 46th international conference on software engineering: Companion proceedings* (*ICSE-companion '24*), April 2024. ACM. https://doi.org/10.1145/3639478.3639806

[43] Julien Malka, Stefano Zacchiroli, and Théo Zimmermann. 2024. Reproducibility of build environments through space and time. In *Proceedings of the 2024 ACM/IEEE 44th international conference on software engineering: New ideas and emerging results* (*ICSE-NIER'24*), 2024. Association for Computing Machinery, New York, NY, USA, 97–101. https://doi.org/10.1145/3639476.3639767

[44] Dmitry Marakasov. 2016-2024. Repology, the packaging hub. Retrieved from https://repology.org/repository/nix_unstable

[45] Simon Marlow and others. 2010. *Haskell 2010 language report*. Retrieved from https://www.haskell.org/definition/haskell2010.pdf

[46] Hannes Mehnert and Louis Gesbert. 2016. *Conex - establishing trust into data repositories*. Retrieved from https://github.com/hannesm/conex-paper/blob/master/paper.pdf

[47] Brandon Milton. 2017. A brief security analysis of arch linux and its package management system. (2017).

[48] Zack Newman. 2021. Security of nixpkgs repository. *NixOS Discourse*. Retrieved November 4, 2023 from https://discourse.nixos.org/t/security-of-nixpkgs-repository/15463

[49] Nix Documentation Team. 2023. Nix 2.19 reference manual. Retrieved from https://nix.dev/manual/nix/2.19

[50] Nix Documentation Team. 2024. Nixpkgs 24.05 reference manual. Retrieved from https://nixos.org/manual/nixpkgs/stable

[51] Nix Documentation Team. 2024. NixOS 24.05 manual. Retrieved from https://nixos.org/manual/nixos/stable

[52] nurmagoz. 2024. NixOS distro preview. *whonix Forums*. Retrieved June 28, 2024 from https://forums.whonix.org/t/nixos-distro-preview/19883

[53] OpenBSD Project. 2020. PROTOCOL.sshsig. Retrieved from https://raw.githubusercontent.com/openssh/openssh-portable/master/PROTOCOL.sshsig

[54] Las Safin and others. 2021. [RFC 0100] sign commits. *GitHub*. Retrieved November 4, 2023 from https://github.com/NixOS/rfcs/pull/100

[55] Justin Samuel and Justin Cappos. 2009. Package managers still vulnerable: How to protect your systems. *;LOGIN* (February 2009). Retrieved from https://www.usenix.org/legacy/publications/login/2009-02/openpdfs/samuel.pdf

[56] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. 2010. Survivable key compromise in software update systems. In *Proceedings of the 17th ACM conference on computer and communications security*, October 2010. ACM. https://doi.org/10.1145/1866307.1866315

[57] Sven Slootweg. 2015. How to deal with package signing? *GitHub*. Retrieved November 4, 2023 from https://github.com/NixOS/nix/issues/613

[58] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. 2017. The first collision for full SHA-1. In *Lecture notes in computer science*. Springer International Publishing, 570–596. https://doi.org/10.1007/978-3-319-63688-7_19

[59] Simon Thompson. 2011. *Haskell: The craft of functional programming* (3rd ed.). Addison Wesley, Harlow.

[60] Santiago Torres-Arias, Anil Kumar Ammula, Reza Curtmola, and Justin Cappos. 2016. On omitting commits and committing omissions: Preventing git metadata tampering that (re)introduces software vulnerabilities. In *25th USENIX security symposium (USENIX security 16)*, August 2016. USENIX Association, Austin, TX, 379–395. Retrieved from https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/torres-arias

[61] Lance R. Vick. 2018. [RFC 0034] Expression Integrity. *GitHub*. Retrieved November 4, 2023 from https://github.com/NixOS/rfcs/pull/34

~