UNIVERSITY OF BERGEN

Department of Information Science and Media Studies

MASTERS THESIS

Reproducible Builds: Break a log, good things come in trees

Author: Morten Linderud
Supervisor: Andreas Lothe Opdahl

June 1, 2019

Simplicity is prerequisite for reliability.

Edsger W. Dijkstra

Abstract

This thesis investigates how transparency log overlays can provide additional security guarantees for rebuilders building Debian packages. In Reproducible Builds it is important to have a set of independent and distributed systems building packages to make sure they have not been tampered with. By putting BUILDINFO files and in-toto link metadata on a proof-of-concept rebuilder transparency log we are capable of detecting tampering of the published logs despite the current scaling problems. This gives users and companies additional security guarantees in the software supply chain for Debian packages.

Acknowledgment

I would like to thank everyone from “Pils og Programmering” for years of moral support, discussions and debates.

My supervisor Andreas Lothe Opdahl for his valuable support and feedback on the thesis.

A big thank you to Santiago Torres-Arias and Lukas Puehringer from Secure Systems Labs at New York University for the opportunity to work on this project and valuable feedback.

I would also like to thank all the students at the Department of Information Science and Media Studies, especially room 642, for time wasted and time spent.

Morten Linderud
June 1, 2019

Contents

List of Figures

List of Tables

List of listings

List of abbreviations

ORM Object-relational Mapping

API Application Programming Interface

SQL Structured Query Language

SDL Software Delivery Lifecycle

SSC Software Supply Chain

REST Representational State Transfer

JSON JavaScript Object Notation

UI User Interface

Chapter 1

Introduction

Distributing software is a difficult task. For years Linux distributions have distributed software by compiling them on build servers, and submitting the packages to a central repository. The packages is then distributed to a number of mirrors where users can get the latest software updates. This enables users to get pre-compiled binaries instead of building them by hand locally. This is an what is commonly referred to as the software supply chain. All the bits and pieces that is involved from writing the code until delivery at the end user.

Targeted attacks against software supply chains is an increasing threat. The security company Symantec reported in 2018 that there was one supply chain attack reported every month in 2017 [12], and in their 2019 report noted that targeted attacks has gone up 78% in 2019 [13]. There has also been a growing concern amongst security professionals about the dangers of supply chain attacks [16], as companies are also struggling to protect their software deliveries [51]. One example is Juniper, a widely used vendor for network equipment, that realized a backdoor was installed on equipment they distributed to customers [10] [36].

Attacks against the supply chains surrounding Linux distributions has also risen the past years. Inappropriate commit access was achieved on the build server used by the Linux distribution Gentoo [26], and the server that holds the repository for the Linux source code [61]. In 2011 the Linux distribution Fedora had one of their contributors targeted in a malicious attack [35], coupled with an incident in 2008 where the build infrastructure was compromised [50]. There have also been attempts at breaking package managers [9].

Packages used in programming languages have in recent years been high profile targets for supply chain attacks. A popular package from the Ruby programming language used by web developers was compromised in 2019, where it started to include a snippet of code allowing remote code execution on the developers machine [60]. What is interesting about this case is that the credentials of the authors where compromised, and the malicious code was never

found in the code repository. It was only available from the downloaded version installed by the Ruby package manager.

Another supply chain attack was in the Javascript ecosystem where a widely used dependency was targeted. The main developer had stopped development, and handed the credentials to another developer who wanted to maintain this library over email. This developer created a new version on Github, but the one uploaded to the package registry used by the javascript language included a code snippet that would attempt to steal crypt currency wallets on the host [49]. The library itself had been downloaded 800,000 times, and is a widely used dependency.

Given the wide reach of supply chain attacks, we need to spend more effort looking at how we can mitigate these risks for users and companies.

In this thesis we will look into how Reproducible Builds enable distributions and software authors to provide bit-for-bit identical binaries. Reproducible Builds is a set of software development practices and guidelines that enables the creation of reproducible and deterministic compilation of software. This has been a large focus by Linux distributions the past few years, and multiple projects has joined the effort to make sure packages and software is reproducible.

We have contributed with Secure Systems Lab at New York University, Reproducible Builds and Debian to provide a system that rebuilds packages to provide additional attestation to the user whether or not a package is reproducible by integrating it into the Debian package manager APT.

The thesis takes this rebuilder system and introduce an append-only rebuilder transparency log, closely resembling certificate transparency logs, where we commit build attestations from the rebuilders. This lets users verify that rebuild attestations have not been tampered with after publication. We will further enhance this log by implementing the possibility of revoking previous build attestations to further help users when verifying packages.

The verification step of downloading packages will be performed by the Debian package manager APT. APT supports the ability to have multiple transports for retrieving packages from the web. To make sure we are able to query the needed services for the verification, we will be providing a new package transport that checks against a transparency log to detect tampering.

1.1 Motivation

The motivation this research is to investigate if transparency logs whether and how this can give the users any new security guarantees on top of the rebuilder verification. There has been no real-world deployment of publicly accessible rebuilder infrastructure, and the integration of transparency logs into user tools has not been investigated

The Reproducible Builds effort and Debian has through the years showed keen interest in providing a secure rebuilder infrastructure, and this project is an important step towards this overall goal of providing this to the end users of the distribution.

It should be noted that this research is primarily focused on the technical implementation of such a system.

1.2 Collaboration with New York University

This project has been done in collaboration with Lukas Puehringer and Santiago Torres-Arias from the Secure Systems Lab department at New York University. This thesis represents the individual research I have done. Together we have built a complete rebuilder system for Debian packages and infrastructure.

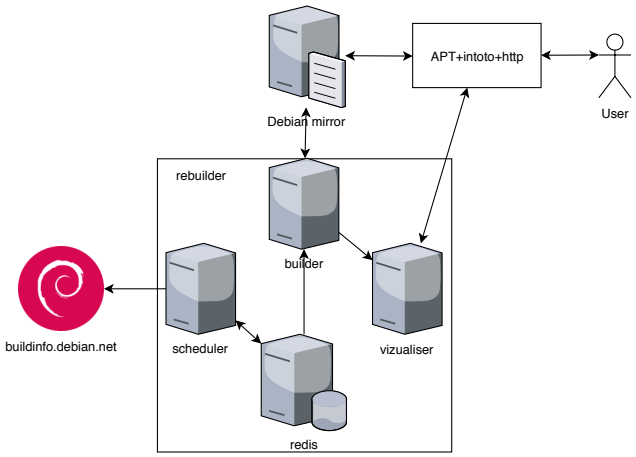


Figure 1.1: Rebuilder architecture overview

The development describes in this thesis is the rewrite of the visualizer component which provides an API to the user clients. The rewrite adds the rebuilder transparency log capability of the system along with providing a more refined API for the APT transport used to verify packages.

1.3 Research questions and contributions

In this section we will present the research question of this thesis. Along with these research questions there is a motivation to contribute reproducible research. All of the evaluations in this thesis have open-source code attached to them.

RQ1

Can a transparency log provide additional security guarantees, and if so, how?

The goal of the project is to see if we are capable of enhancing the visualizer component of the rebuilder with more security guarantees. Transparency logs can provide security gurantees if implemented correctly, namely the evidence that the provided logs have or have not been tampered with. The current rebuilder has no such feature and any build submissions can be tampered with after publication.

RQ2

Are we able to implement this into the current rebuilder verification process?

The current rebuilder verification process fetches plain text data from an endpoint with no validation. To utilize the security features from a transparency log, we would need to make sure they can be validated and implemented in the APT package manager.

RQ3

Can this be deployed in a real-world scenario?

Given a correct implementation of the transparency log, it would be interesting to investigate how the resulting log implementation can work. Debian publishes multiple packages each day, and we can see the amount of data the log would need to consume and whether or not it is capable of consuming the data in a real-world scenario.

Chapter 2

Theory

In this section we will take a closer look at the theory surrounding supply chains, reproducible builds, transparency logs and rebuilders.

2.1 The Supply Chain

Most of the software development today are developed through a series of steps. This is traditionally called “The Software Supply Chain”. Software projects today go through development, building, testing, staging and production, with slight variations between them. This is largely done with self-hosted solutions, or outsourced to external hosting solution.

In the world of open-source, the software delivery are usually done by Linux distributions, or other similar methods of distributions. The supply chain in this regard is the complete steps from developers getting the source code for the project, until it is delivered as a compiled artifact to the end-user. This also includes the wider network of packages that are needed for the distribution of the project. To get understand the wider problem of delivering secure software, we will contrast “Software Supply Chain” with the more commonly thought of “Software Delivery Lifecycle” to understand what the supply chain encompasses.

Lipner in “The Trustworthy Computing Security Development Lifecycle” outlines the “Software Delivery Lifecycle” as a development model on how to deliver secure software [40]. This is done with the following steps;

- Requirements
- Design
- Implementation

- Verification
- Release
- Deployment
- Response

For each of these steps there are adequate security measures assigned. The “Requirements” step would need the developers to assess the security requirements of the process, and to make sure any milestones are met. The “Release” step would for instance include a penetration testing, where an active party attempts to hack or compromise the given software, and have a threat model reviewed where the security concerns are addressed and in some case justified. This model only encompasses the development and the code written by the authors and is fairly similar to a traditional software models.

Traditional software models like Agile development, waterfall and extreme programming is all about managing the development lifecycle. Write code, respond to changes and delivery a product. However, we are lacking a few considerations from this model; distribution of the software and the wider ecosystem that is involved writing and producing software. Ellison in “Evaluating and Mitigating Software Supply Chain Security Risks” he analyzes how the United states Department of Defense handles software acquisition in a very high secure environment. In the context of the military “[...] supply chains typically involve the movement of materials from home base to troops in theater. The responsibility for managing these supply chains falls to the acquisition and logistics experts” [22]. In this case, the DoD is not producing any software. Their only concern is to get the software developed, tested, shipped and updated in a secure fashion. In all of these steps they might relay on outside contractors and thus have to safeguard themselves against any risks. This in turns makes the supply chain far larger, and far more encompassing then a development model like the software lifecycle looked upon earlier.

In this thesis we will focus on how package managers work as a supply chain for distributing software in a secure manner.

2.2 Linux Distributions

Linux

Linux is a free and open-source kernel. It was first developed by Linus Torvalds in the early 1991 and has grown into the largest open-source project today [25]. It is commonly used in

everything from firmware modules on a computer, to the ever increasing field of Internet of Things, along with servers and on personal computers. The development of Linux is distributed and has spawned the open-source method of developing software.

Linux is accompanied by a suite of tools and environment that is commonly referred to as a distribution and defines an operating system based on Linux. These are created by companies as commercial products, as well as groups of volunteers as a hobby for free. The tooling of these distributions, along with organization and the inherent supply chain to deliver artifacts to the users, are unique to each project. Some are "source"-distributions, and only distribute build recipes, and some distribute pre-compiled binary packages.

Debian

Debian was one of the first operating systems based on Linux, and was created by Ian Murdock in 1993. One of the main innovations from Debian was the creation of the very first package manager [18]. Package manager allows users to download pre-compiled software from centralized repositories maintained by the Debian developers. This allows users to easily fetch, update and remove installed packages on their system.

These packages are maintained by package maintainers who package, update and maintain the required files to distribute the packages to the end user. Each maintainer has a cryptographic secure signing key they use to fetch, and publish source packages to a build server. These source packages contains the needed files, patches and package files to compile the project source code to a format accepted by the Debian package manager, APT, which allows the users to easily install and remove software.

The build server verifies the signatures and compile these source packages to all the supported architectures. It will then sign these packages with its own key, and distribute them to the end-user in the form of a mirror system.

2.3 Software determinism

Supply chains can be complicated and ensuring that all parts of it is secured can be hard work. How can we make sure the packages from these supply chains are not tampered with? Modern toolchains that aids in building software can be complicated and embed a lot of information which might not be present in future builds. We will be taking a look at some of the fundamental work that has gone into considering undeterminism in software and also look at the Reproducible Builds effort which is a software development model to aid in producing deterministic

software.

Trusting trust

In a paper from Ken Thompson in 1984 (after he won the Turing Award for his work on the UNIX operating system) "Reflections on Trusting Trust", Thompson as a programming exercise implements a very basic self-reproducing program. As a demonstration, he adds code capable of introducing new code when certain patterns are encountered. This open up the possibility of having malicious code inserted into compiled code that could be leveraged by malicious actors.

> "You can't trust code that you did not totally create yourself. [...] No amount of source-level verification or scrutiny will protect you from using untrusted code. In demonstrating the possibility of this kind of attack, I picked on the C compiler. I could have picked on any program-handling program such as an assembler, a loader, or even hardware microcode. As the level of program gets lower, these bugs will be harder and harder to detect. A well-installed microcode bug will be almost impossible to detect"[63].

This paints a very bleak picture, considering most software we get today is pre-compiled and provided to us by different vendors. There is no clear cut of verifying what is provided by these vendors. But there are possible ways to counter this problem. David A. Wheeler in his dissertation "Countering Trusting Trust Through Diverse Double-Compiling" details a possible solution to the "trusting trust"-problem. It involves what he call "diverse double-compilation" [66].

"Diverse double-compilation", or "DCC", is the act of using two compilers to detect any difference in the resulting artifact. The first compilation is done with a secondary compiler, then again with the primary compiler. The idea is that the secondary compiler is a minimal implementation of the compiler, and can be trusted. However, creating compilers is not a trivial task. If we want to have a trusted and verified compiler that is capable of outputting the same binary we need to write these compiler our self. This gets complicated quickly when you consider the time and effort spent on writing compilers for languages such as C++. Instead we have made an effort the past years to reproduce deterministic software in other ways.

Reproducible builds

Reproducible builds is a set of practices on how to achieve deterministic compilation of software. Supply chains are usually handled with multiple tools, and on several individual computers. This leaves a rather large attack surface for malicious actors to try compromise the chain.

This is not a theoretical threat. There has been an increase in attacks on parts of supply chains in recent years. They are high impact and affect users as well as developers.

According to the definition of reproducible builds;

> "A build is reproducible if given the same source code, build environment and build instructions, any party can recreate bit-by-bit identical copies of all specified artifacts" [2].

One of the earliest open-source projects to promote reproducible builds is the Tor project. The Tor project develops the "Tor network" which is an anonymity network compromised of volunteers that run network nodes that effectively anonymize the network of the user [20]. This is heavily used by dissidents in oppressive regimes where the internet connection is filtered, or partially blocked. To help giving access to this network, they develop a variant of the web browser Firefox called "Tor browser" [62].

The browser is configured to utilize the Tor network for anonymous internet browsing. Marginalized protesters has been widely using Tor to circumvent censorship. The supply chain and distribution of this software is important as protesters and marginalized groups need to make sure the browser is not compromised. Receiving a malicious version of this software could in many cases result in prison or life threatening danger to people. Mike Perry highlighted the concern when discussing the testing of the "Tor browser".

> "[...] software development has to evolve beyond the simple models of "Trust my gpg-signed apt archive from my trusted build machine", or even projects like Debian going to end up distributing state-sponsored malware in short order" [53].

The result of this concern is the move to support reproducible builds: Allowing users, and multiple independent builders, to recreate the distributed artifact bit-for-bit [42]. This lets the user themselves compile the Tor browser and make sure there has been no tampering with the distributed binaries if there was a suspicion they where. This is done by utilizing Gitian which builds and packages the software on self-contained virtual machines [46]. This enables the project to distribute the same build instructions as used to package the software in the first place, and allows users to easily verify if the distributed artifact matches the self-produced one.

Around the same time as Gitian was discussed, in 2013, Debian started a push towards reproducible builds, and an effort into achieving this for their distributed packages [41]. Since then, 22 projects are officially part of the initiative to bring reproducible builds to users, among others are Arch Linux, Fedora, Tor, openSUSE and more [7]. There has been 4 summits held for volunteers to come together and discuss the issues at hand [3].

Source Date Epoch

One of the most common offenders for undeterministic builds is the embedding of when something was built. On the surface this looks like a completely innocent thing to do for most builds, but it creates problems when the produced artifact in turns become undeterministic because we built it at another point in time.

The Reproducible Builds project defines an environment variable called "SOURCE_DATE_EPOCH" which is a means to solve this dilemma [6]. This variable enables software distributors to build artifacts with an embedded time, but it also helps to specify the time in a manner that enables reproducible builds. We can record this variable, and at a later point in time define the variable to pretend we build this at the given time.

The requirement is that this variable is exported to the build system used to create the package. It also needs to take the current date time if no "SOURCE_DATE_EPOCH" is provided.

Buildinfo

One of the main issues with reproducible builds is that it is hard to make everything universally reproducible. Producing the same binary package on multiple different Linux distributions is close to impossible and unmanageable for most software. Thus we need to specify the environment being utilized with all of the requirements and idiosyncrasies.

The environment is made up of a few things. The installed software installed on the system. What shell variables are present as it denotes the expected timezone, language settings and any other special requirements. These things can make builds behave differently and are thus important to record and keep track of.

This is not a new discovery. Cabrera and Appleton in their paper "Software Reconstruction: Patterns for Reproducing Software Builds" from 1999, defined a "Bill of Material", or a "BOM" for short.

> "Document all of the components that contributed to the build inn a list, i.e., a bill of materials (BOM). The BOM may contain the names, versions, and directory paths of operating systems, libraries, compilers, linkers, make-files, build scripts, etc The BOM may be manually created, but many configuration management tools generate it as a byproduct of the build" [8].

```
Format: 1.0
Source: dh-make
Binary: dh-make
Architecture: all
Version: 2.201802
Checksums-Sha256:
22c95094efbe79445336007dd[...] 42360 dh-make_2.201802_all.deb
Build-Origin: Debian
Build-Architecture: amd64
Build-Kernel-Version: 4.9.0-8-amd64 #1 SMP Debian 4.9.110-3 (2018-10-08)
Build-Date: Thu, 06 Dec 2018 00:04:23 +0000
Build-Path: /build/dh-make-2.201802
Installed-Build-Depends:
autoconf (= 2.69-11),
automake (= 1:1.16.1-4),
[...]
xz-utils (= 5.2.2-1.3),
zlib1g (= 1:1.2.11.dfsg-1)
Environment:
DEB_BUILD_OPTIONS="buildinfo=+all reproducible=+all parallel=16"
LANG="C"
LC_ALL="POSIX"
SOURCE_DATE_EPOCH="1543231660"
```

Listing 1: Example BUILDINFO file from Debian

Such a file would contain all the requirements to recreate the environment the artifact was built inn. Because each ecosystem has their own way of parsing information, there is a need for multiple formats to define the requirements. Currently there are around 5 different formats for different ecosystems on the reproducible builds website [4].

Listing 1 shows the format used by Debian. It encompasses a wide array of values. “Format” defines the expected fields in the format and gets incremented with any changes. “Source” and “Binary” defines the source package used to produce the artifact, and the corresponding binary package it outputs. The “Architecture” fields defines which architecture the product is compiled towards. Debian supports a wide array of CPU architectures from ARM to AMD 64 bit. High-level languages usually does not compile, therefore “all” is used to denote the architecture [19].

The “Build-” variables denote the build environment used to create the artifact in the Debian ecosystem. Since Debian has a slew of derivative and closely related distributions, “Build-Origin” is used to denote this distribution. “Build-Architecture” denotes the architecture of the build server being used. “Build-Kernel-Version” denotes the explicit version of the kernel used.

This is commonly fetched with the command “uname -a”. “Build-Date” refers to the ISO compatible date when the process took place. “Build-Path” is the location where the build took place. “Installed-Build-Depends” contains list of all packages present during the packaging of this artefact. This s an important list to keep track of as it enables the complete recreation of the environment at a later point.

Linux has several variables that set the locale, language and timestamp format that can affect the build process. “Environment” encompasses all of these. Most importantly the variable “SOURCE_DATE_EPOCH” is stored here to make sure timestamps are deterministic.

This file can then be distributed alongside the package, or provided through other means. Debian archives all buildinfo-files on a centralized webpage where they can be queried and retrieved [37].

When distributing files it is very common to do so using file archives. Common formats are the ZIP and the TAR format. The order in which files appear in the archive needs to be consistent for the checksum of the archive to match, however this can be hard to test in some cases. “disorderfs” is a filesystem that lets you introduce unexpected and randomized behavior when reading files. This helps find sources for non-deterministic artifacts in the build process.

Diffoscope is a tool to help compare binary formats for differences. It supports a lot of binary formats to help find reproducability issues in produced artifacts. It enables the user to output reports of the comparison as plain text files or HTML files so they can be easily embedded in webpages. The current CI system in Debian provides the HTML files for easier debugging.

The software is packaged and used by a number of distributions and is currently a very important tool in debugging reproducability issues.

2.4 Rebuilders

One of the main ideals with reproducible builds is the ability to let the users recreate distributed artifacts. This is achievable with the correct tooling, and a “BUILDINFO” file as specified in Listing 1 on page 11. However, building all distributed packages is an unwieldy task. The appeal of Linux distributions is the ability to download pre-compiled packages to save the effort of building all the software one intends to use.

Debian has put a great deal of effort into testing packages by setting up a “Continuous Integration” framework for testing packages [31]. This setup compiles all Debian packages twice with variations to see if they end up reproducible or not. This is a neat approach to find reproducability issues, but it does not reproduce any packages produced and distributed by Debian

developers directly. They are merely built twice in their own environment with the packaging files distributed by the developers. The real goal is to reproduce distributed packages, so the CI solution it self does not fulfill this goal.

What is needed are servers which take the same packages which are distributed, and reproduces these. The general idea is that there should be a pool of diverse servers which are capable of rebuilding packages in the correct environment [5]. These rebuilders should be capable of sharing build attestations of the built package. The build attestations used in this project is the BUILDINFO files, described in Section 2.3 on page 10, and the in-toto link metadata described in Section 3 on page 22. The in-toto link metadata files will be used to verify the downloaded package on the user system.

2.5 Rebuilder logs

The rebuilder system needs to publish build attestation and the results of the package builds it performs. The files published is the in-toto link metadata used for package verification, and BUILDINFO files which describes the build environment. Since these are log files, they could be tampered with and altered to pretend a malicious package build is in fact correct. What we need is evidence whether or not the logs has been tampered with. This can be accomplished by utilizing something called transparency logs. These data structures are built on Merkle trees, where nodes are hashed together to construct a binary tree, which can prove whether or not a published log has been tampered with. In this section we will take a look at how such logs work and the theory behind them.

2.5.1 Merkle Trees

Merkle Trees is a tree structure based on cryptographic secure hashing function [45]. It creates a binary tree where each leaf is hashed together to create interior nodes. The top node of this tree is referred to as a root node and the data structure is essentially a binary tree structure where two nodes has two children. The main idea behind this scheme is to provide a cryptographic signature that can consist of multiple messages together. But Merkle tree has most widely been used for creating tamper-evident logs.

Transparency logs

Transparency logs is a use case of Merkle trees by Crosby and Wallach which describes how this data structure can be used to achieve tamper evident logging [15]. In the traditional sense, logs

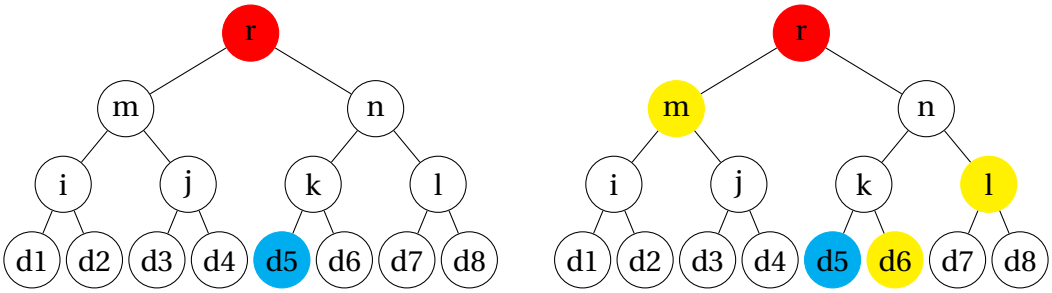
can be plain text files published anywhere. It can simply be a line of text detailing an event. If this where to change on a remote service, there would not normally be any evidence of this. What a transparency log attempts to accomplish is to create hashes of the log entries, and create a tree. If we store the hash of this tree, we could prove the log has been modified at some point in time after the has we got was provided.

Certificate transparency logs from Laurie, Langley, and Kaster is an implementation of transparency logs. These logs lets organizations that issue HTTPS certificates an audit trail [38] when issuing SSL certificates. The audit trail allows the discovery of when abuse, and creation of malicious certificates. The log allows organizations to see who issued certificates for what domains, and can aid in detecting private key compromise, API misuse or other types of malicious usage early.

To make sure we can prove evidence of tampering, transparency logs has the unique feature where we can provide some hashes from the tree, and reconstruct an assumed root node. Given a sound and secure hashing function, difference in the root tree checksum proves if modifications has been done on the tree. The proofs needed to implement a transparency log is as follows;

- Audit proof
- Consistency proof

Proofs are tuple pairs where the first element describes the position of the hash, which can be either left or right, and the second element includes the hash of the given object. Given the correct order of hashing, the product should be some Merkle tree root the log is either currently using, or have used in the past. If this is not the correct root hash it proves some tampering has been done on the log.



(a) Merkle tree with leaf “d5” highlighted

(b) needed nodes for audit proof of “d5”

Figure 2.1: Audit proof

The audit proof is used to verify that the given element in the log has not been tampered with. The tree root is given, along with the elements needed to recreate the missing nodes for the root. In the Figure 2.1 we can see the representation of a Merkle tree. The leafs in this tree $d1$, $d2$, $d3$,... $d8$ are hashes of the log values we are storing on the tree. In our visualizer components this is the rebuild submissions consisting of the BUILDINFO file and in-toto link metadata.

For the proof that the node $d5$ is part of the log we need to reconstruct the tree root r. It's important to realize that hashes can be a shortcut when reconstructing the tree. We only need the tree of the largest subtree the leaf is not a part of. Since we want to check $d5$, the leafs $d7$, $d8$ are not needed, we can just utilize the complete subtree l which is the hash of both those nodes. This ensures that we only need the minimal amount of hashes to reconstruct the tree root r.

To reconstruct the tree root, the tree nodes we need are $r = \{d5, d6, l, m\}$. If they are hashed together appropriately, such as $r = H(m, H(H(d5, d6), l))$ where H defines a secure hashing function such as SHA256 or SHA512, we will arrive at the same value for and thus the proof is validated.

The consistency proof is used to verify that the log is operating as an append-only log in a correct manner. This proof requires two things. A previous Merkle tree root, and the number of leaves present at the time of this tree root. The returned path is the number of subroots needed to recreate the path from the root, until the new root.

To have a correctly vetted tamper-evident log, there needs to be monitor server. Monitor servers watch new entries from log servers and verify them. They collect the current signed tree root, and make sure they match up with the transparency log by validating consistency proofs. These can be run by volunteers or organizations to make sure the logs are behaving correctly.

Transparency Log Overlays

For our system we will add some semantic meaning to the leaves we add to the Merkle tree. This will be explained in greater detail in the development of this in Section 5.2.4 on page 58, however we will go into some of the underlying concepts.

Weippl et al. describes in "Transparency Overlays and Applications" a concept of adding transparencies on top of existing processes to achieve transparency and tamper-evident events [65]. They create a system greatly inspired by the work of both Crosby and Wallach, in transparency logs [15], and Laurie, Langley, and Kaster in certificate transparency logs [38], and builds on top of of the concepts describe in the papers.

The idea of this approach is to log the everyday events of systems on a transparency log to

make sure the events are tamper-evident. The values of these commitments on the log can be arbitrary and aid in providing some semantic meaning to the underlying application. In our project, we will utilize this idea to provide rebuild attestations, but also the ability to revoke such attestations on the Merkle tree.

2.6 Research Overview

In this section we will be taking a look at the work done in the surrounding areas when it comes to reproducible builds, supply chain security and transparency logs.

2.6.1 Software Distribution Transparency and Auditability

The initial work on package transparency logs, as a form of binary transparency, was done by Hof and Carle. It implements an append-only Merkle tree to keep track of released versions of software into a package repository, namely Debian [29].

The paper details a novel hidden attack, where a backdoored package is distributed to some users but not everyone. As the transparency log knows which package is the correct one, it is possible to detect such attacks. The implementation also defines log monitors, that peek at inclusions and make sure the logs are operating properly and doesn't leave out information.

The usage of transparency logs in this project inspired the work done in this thesis. The idea here is that the transparency log by Hof and Carle can be applied as release monitor and detect malicious or compromised signing keys by developers. This is an addition to the rebuilder system described in this thesis, and the goal is to see if the addition of transparency logs on the rebuilder gives us any security guarantees.

2.6.2 CHAINIAC

CHAINIAC by Niktin introduces a framework to help collectively validate source-to-binary correspondence[48]. This is done by introducing a cothority. Each developer commits their binaries and corresponding source code to a Merkle tree which is then signed by the developers. This is built through a distributed system which will rebuild and corroborate their results to aggregate signatures on packages which makes up the cothority. The number of required signatures from this cothority is defined by the project and pushed to an updated timeline.

The main problem with this project is that it imposes new requirements on the distribution developers. Changing how distribution developers work with packages, and how packages are

released is something that has been done for years. Changing this approach is not going to help adoption by the distribution. The rebuilder system in this thesis is an extension of the current release process, it does not impose any new methods or requirements on the developers.

2.6.3 Contour

Contour by Al-Bassam and Meiklejohn is a system that implements binary transparency by utilizing blockchains. In blockchains Merkle trees are used to store the data. In this implementation the resulting binaries are hashes on the chain. The resulting Merkle root is then hashed with previous roots to provide the transaction and a block header. This is again distributed and secured from split view attacks, where one server presents the client with a malicious view of the world. Split view attacks enable the remote log to provide proofs to the client which are not present elsewhere. The solution to this is by adding a consensus on top of the tree. The implementation was tested using the Python packaging index, PyPI, and the Debian package repository to test the solution [1].

This project attempts to commit the Debian package index to a Bitcoin blockchain to make sure the index stays consistent across the ledger. This is an integration which is strictly not needed to verify packages. Our research project presents a simpler solution where consistency across rebuilders are not needed.

2.6.4 Go transparency log

Go has been developing their dependency management the past year. With the release of their new dependency manager, go mod, they now produce dependency information along with a lockfile. The lockfile, go.sum, contains packages along with versions and checksums.

A recent proposal by Cox and Valsorda details how this go.sum file would be commited to a transparency log to provide a verification method for dependency releases in the go ecosystem. It implements a own database to model Merkle trees, and lets nodes validate and monitor the tree for dependency inclusions. The idea is to check this log server when dependency files are downloaded for proofs of releases.[14]

This project only appends the project dependencies as a way to pin it, and enables developers to make sure they have received the proper dependency listing of the project.

Summary

In this chapter we have taken a deeper look at the theory surrounding software distributions, supply chains, reproducible builds and Merkle trees. The supply chain is important to secure, and Reproducible Builds is possibly one way to achieve deterministic building in the world of Linux distributions.

We have looked at how rebuilders can help verify packages are reproducible and have introduced transparency log as a way to make sure the logging of these package builds can not be tampered with after they have been published.

In the next chapter we will be taking a look at the technologies we will be utilizing for this project.

Chapter 3

Technologies

In this section we will take a look at the technology chosen for this project.

Python

Python is a general purpose programming language created by Gudio van Rossum in 1994 [55]. It's dynamically typed language, with a terse syntax and a wide selection of built-in libraries for developers. Dynamically typed languages lends itself nicely for rapid prototyping of experimental projects. A language like Rust or Go, which are statically typed gives the developers some more issues prototyping data structures as all types needs to be consistent and decided up on early. With Go this becomes a worse problem with the lack of generics.

Python also has a wide selection of well maintained and frequently used libraries that we can utilize for our project. Python is commonly used for back-end development and web services inn general and has good libraries for this.

Transparency Log

The development of the transparency log requires some effort to properly implement a Merkle tree with the appropriate proofs to so the implementation can be useful. To do this we lifted some code from the Python library "pymerkle" by Foteinos Mergoupis to aid in the development [24]. Parts of the algorithms has been changed to accommodate the development goals of this project. It was easier to lift code then find a suitable merkle tree library to integrate with to provide the needed features.

flask

Flask is a web framework for Python. It was created in 2010 by Armin Ronacher, and is one of the two most widely popular web frameworks in Python. It enabled developers to easily create REST API endpoints with good debugging capabilities [56]. This framework will be used to create the API endpoints for the rebuilder.

jinja2

One of the added features of using flask is that we get access to the templating framework jinja2 which allows us to easily create webpages and interface them with Python values before being served to users. This allows us to easily create webpages with the data used by the application [57]. We will be utilizing this library to create the HTML webpages for the rebuilder.

PostgreSQL

PostgreSQL is a open-source relational database. It supports a wide number of abstract datatypes, such as native support for JSON, along with good support for concurrent operation. This enables easier development, along higher workloads and scalability [54]. We decided to utilize PostgreSQL over other technologies such as MySQL or sqlite because of the ability to embed JSON structure directly into the code.

SQLAlchemy

SQLAlchemy is a widely used object relation mapping library for Python. It support a wide selection of database backends and translates the raw database data into usable Python objects for easier interopability. This helps us to save time by not having to map any values to our own data structures as everything are written as native Python classes [59].

Listing 2 shows an example ORM model where we define a user table. Each model has a created timestamp which defaults to the current time, and a name value which defines a string for the name

Panda and matplotlib

For graphing we will be using pandas and matplotlib. Both used a lot when it comes to scientific computing with Python. It enables powerful graphing capabilities over simply dataformats,

```
class User(db.Model):
    __tablename__ = "user"
    id = db.Column(db.Integer(), primary_key=True, autoincrement=True)
    created = db.Column(db.DateTime, default=datetime.utcnow)
    name = db.Column(db.String(96), index=True)

    def __repr__(self):
        return "<User: {}>".format(self.name)
```

Listing 2: Example SQLAlchemy model

such as CSV, and are tightly integrated into one another [43][32]. We will be utilizing these libraries to generate graphs during the evaluation phase on this thesis.

Git and Github

An important aspect of any project is to keep track of changes. For this project the version control system "git" was used. It is a well known version control system and widely used and deployed on a wide selection of providers. The strongest point is being decentralized, where commits and code changes can be done without internet access [39]. The code can also be pushed to multiple repositories for backup purposes if any provider goes down during the development of the project.

Docker and containers

One of the goals of this project is to make sure the testing is done in a reproducible manner. To do this we need to make sure the environments are consistent between testing, and that they can be reproducible rebuilt after the testing is complete and into the future. To do this the development is done in containers, namely a technology called Docker [44]. Docker enables reproducible containers with a given Linux distribution and dependencies. They can be versioned and recreate environments across machines.

We will be utilizing this technology for extended testing for deployment and data structure testing in this thesis. We will be providing scripts to recreate the environments used to run the evaluations with docker and the container technology.

curl

When developing an API is desirable to quickly test the functionality. For this project we utilized the widely used and popular tool curl, which is written by Daniel Stenberg and supports a wide array of URLs and data protocols for data transfer [17]. This allows us to quickly and effectively test and display API end points we are going to be developing.

in-toto

in-toto is a framework to verify the integrity of a supply chain. It defines a specification that details what steps should occur. As one supply chain could define and utilize any number of steps its vital for this to be extensible, and customize able. In-toto lets the specification detail who should perform the step in the supply chain [58].

The layout describes what each step of the supply chain should contain. It can be any expected commands, something the process should succeed running, any expected material, things needed for the step in the chain to proceed, and any products, artifacts created by the steps. These are described in a very small language that contains keywords following regular expressions which should be satisfied.

The link metadata is a JSON file that specifies what the values, and outputs of the corresponding step should be. Evaluating the specification along with the link metadata lets the users, or the organization, verify that the supply chain has not been tampered with.

```
{"signatures": [],
 "signed": {
  "_type": "layout",
  "expires": "2021-01-06T18:30:57Z",
  "inspect": [{
        "_type": "inspection",
        "expected_materials": [
             ["MATCH", "*.deb", "WITH", "PRODUCTS", "FROM", "rebuild"],
             ["DISALLOW", "*.deb"]],
        "expected_products": [],
        "name": "verify-reprobuilds",
        "run": ["/usr/bin/true"]}],
  "keys": {
        "2e7be98291270e3b7fca429a2210e99cff22017e":{
             "hashes": ["pgp+SHA2"],
             "keyid": "2e7be98291270e3b7fca429a2210e99cff22017e",
             "keyval": {"private": "", "public": {"e": "010001", "n": "e0da84bec..."}},
             "method": "pgp+rsa-pkcsv1.5",
             "type": "rsa"}},
  "steps": [{
        "_type": "step",
        "expected_products": [
             ["CREATE", "*.deb"],
             ["DISALLOW", "*.deb"]],
        "name": "rebuild",
        "pubkeys": ["2e7be98291270e3b7fca429a2210e99cff22017e"],
        "threshold": 1}]}}
```

Listing 3: Example in-toto schema

```
{"signatures": [
  {"keyid": "918b19596...",
   "other_headers": "0400010800...",
   "signature": "bc1d9776bf..."}],
 "signed": {
  "_type": "link",
  "name": "rebuild",
  "products": {
   "python-sshpubkeys_3.1.0-1_all.deb": {"sha256": "8e69d5cbdc..."},
   "python3-sshpubkeys_3.1.0-1_all.deb": {"sha256": "8234484139..."}}
}}
```

Listing 4: Example linkmetadata file

Listing 3 shows a in-toto schema. This is utilized to show the needed steps in the supply chain. It aids in verifying who can sign off on each step, what keys are used and what the acceptable output can be. To verify this we need a link metadata which is an output of the supply chain. Listing 4 is an example metadata file that contains a signed layout with the products and checksums from the product. These two files together allows us to verify that each step of a given supply chain has been don.e

In this project we will be utilizing in-toto for the build attestation verification part in the APT transport. In the in-toto APT transport we provide a in-toto schema where the rebuilder provides the appropriate linkmetadata. Each package downloaded by APT gets verified by running the in-toto process over these two files.

Deployment

UH-IaaS is a cloud infrastructure provider created for research institution to deploy software and build platform [33]. They provide free resources for student projects and the rebuilder infrastructure described in this thesis is currently deployed on their platform. Extended testing of the implementations of this thesis is also done on their platform.

Summary

In this chapter we have taken a deeper look at the theory surrounding supply chains, reproducible builds and Merkle trees. In the next chapter we will be taking a look at the current research being done towards shared attestation on reproducible builds and transparency logs.

Chapter 4

Research Method

This chapter goes through the research methodology, methods and the chosen framework used in the research. We will address the research questions presented in Section 1.3.

4.1 Design Science Research

Design science research is a research methodology where the aim is to design and develop Information Systems and IT applications. For our project we are creating new behavior and need to evaluate them, this fits nearly with the goal of Design Science, as Hevner et al. writes:

> "... design-science paradigm has its roots in engineering and the sciences of the artificial. [...] It seeks to create innovations that define the ideas, practices, technical capabilities, and products through which the analysis,design, implementation, management, and use of information systems can be and efficiently accomplished" [28].

In contrast to empirical science, where nature is observed and tested to understand the observation and gain knowledge, design science is about identifying the need for something and attempting to create a solution to this problem. The following evaluation of how well this solution fits the problem at hand can give us valuable research in the field of information science and computer engineering.

> "The main research activities involving the natural sciences are to discover how things are and to justify the reasons for them being so. Natural science research should be faithful to the observed facts while also being capable of predicting future observations to some degree" [21].

The result of this research should be an artifact which contributed something to the field. Vaishnavi and Kuechler listed a few examples of potential contributions Design Science can result in [64]:

- Constructs - The conceptual vocabulary of a domain.
- Models - Set of propositions or statements expressing relationship between constructs.
- Frameworks - Real of conceptual guides to serve as support or guide.
- Architectures - High-level structures of systems.
- Design principles - Core principals and concepts to guide design.
- Methods - Sets of steps used to perform tasks.
- Instantiations - Situated implementations in certain environments that do or do not operationalize constructs, models, methods, and other abstract artifacts; in the latter case such knowledge remains tacit.
- Design theories - A prescriptive set of statements on how to do something to achieve a certain objective

In this project the main artifact is an reimplementation of the visualizer component, as described in 1.2, with additional improvements. Following the list from Vaishnavi and Kuechler this is an “instantiation”.

4.2 Guidelines

Hevner et al. argues that design science is inherently a problem solving process. We need to understand the problem, build a knowledge base and then try come up with a solution to this problem. To aid in this, they defined a set of seven guidelines to assist researches in constructive proper design science research projects [28].

Design as an artifact

The main goal of design science research is to produce an artifact. They are incomplete or complete projects, guidelines or insight into the problem they are trying to solve [28]. Example of different artifacts as described by Vaishnavi and Kuechler can be new methods for developing software, new systems for processing data called Instantiations or new design principles for writing software [64].

Problem relevance

The artifact should be relevant to the problem it is trying to solve. Hevner et al. argues that problems can formally defined as the difference between the end goal of a state, or and the current state of things [28]. Bridging this gap needs an understanding of the current problem domain to achieve the needed artifact.

Design evaluation

The artifact needs to be tested and evaluated to figure out how well it solves the problem at hand [28]. This is important to provide rigor in the research. It also provides assurance of the high quality the artifact is supposed to have.

Research contributions

The resulting design science research should result in a contribution to the research field [28]. Hevner et al. says one of the following contributions need to be founds in a research project;

1. The Design Artifact
2. Foundation
3. Methodologies

In our case the artifact is an“Instantiation”, thus the “Design Artifact” is the research contribution. The end goal is to take two separate concepts; the rebuilder and the transparency log, and attempt to combine them to see if we gain any additional security guarantees.

Research rigor

Hevner et al. says that “rigor is achieved by appropriately applying existing foundations and methodologies” [28]. This research is based on seminal work done by previous research, along with the important work done by free- and open-Source projects in this field.

Design as a search process

Design science is suppose to be an iterative process. The goal is to find effective solutions to the problem at hand. This is done by iterating on previous implementations until something that

solves the problem is reached [28]. Hevner et al. writes that solving problems needs means to reach the desired ends while satisfying laws from the environment.

Communication of research

The resulting research needs to be clear, understandable and concise. One should be able to presented the research to both technologically-inclined people, as well as people in the business setting. This is important to convey the importance of the research, as well as aiding future researchers to build off on the research done in this project [28].

4.3 Evaluation

Evaluation is a part of design science research. Because our artifact is an Instantiation we need to decide on a way to test the actual code base being produced in this thesis.

Peffers et al., in "Design Science Research Evaluation", does a literature review on 148 design science articles. They do an open coding approach, where they categorize the articles using codes, and compare the results across the different fields [52]. Their review isn't very detailed, as noted in their conclusion, but it pinpoints some trends among engineering fields where design science artifacts, such as instantiations, are usually evaluated with technical experiments.

They explain technical experiments as;

> "...a performance evaluation of an algorithm implementation using real-world data, synthetic data, or no data, designed to evaluate the technical performance, rather than its performance in relation to the real world. " [52, p. 402]

To evaluate our artifact, we are essentially going to take a look at the package builds debian does, and see if we are capable of processing them with our system. This gives us some real world data to take a look at, and help us figure out if we are on the correct track regarding our system. We will also be writing up a API endpoints.

An API are important as they are utilized by developers. However, there are no clear process on how to evaluate good API design. Iyer and Wyner outlines this problem in "Evaluating APIs: A Call for Design Science Research", and comes up with a stakeholder analysis to help evaluate API design choices when it comes to design science [34].

More importantly, they detail a set of key attributes which they believe are important for good modular API designs. The outline of these key trais are what we are also going to take a look at [34, p. 31];

Functionality - The modules are separated and contain logical groupings.

Hierarchy - Modules can be decomposed into sub-modules and internal details does not leak.

Separation of concerns - Each module is loosely coupled with other modules.

Interoperability - Modules can easily interact with others.

Resuability - Modules can be reused in other systems.

This coupled with a real-world evaluation when it comes to data will guide us in the evaluation stage of this research.

Chapter 5

Development

In this chapter we will be taking a look at the development in this thesis. The first section will be focused on learning what the current rebuilder setup is doing, establish the base requirements of the rewrite of the visualizer with the goal of improving the design.

After the first iteration is done we will be implementing the Merkle tree storage for the rebuild submissions which consists of a in-toto link metadata file, and a BUILDINFO file. Then we will be focusing on implementing the needed signing of the Merkle tree roots to help verify that the given tree roots actually comes from the given rebuilder. The final iteration will be the implementation of the transparency log overlay, which will be the front end for the APT client to fetch rebuild submission and check if they have been revoked or not.

The last section deals with the integration between the existing APT transport and the new visualizer API. We will be removing the support for the old API and connect it with the new rewritten API with support for transparency logs. We will then make sure the test suite for the transport completes and everything works.

5.1 Rebuilder

The purpose of the rebuilder is to watch for new packages, queue them, build the package in a clean environment to reproduce the package, and then publish this result so we can query these later when installing packages.

To achieve this we need to fulfill a few requirements:

1. We need to know when a package is published.
2. Something needs to schedule the new packages.

3. We need to build the package in a clean environment.

4. We need to publish results of the built package.

5. We need to check the results when installing packages.

It is important to remember that this system is only targeted at Debian as supporting it universally would require a lot of engineering effort and handling of special cases.

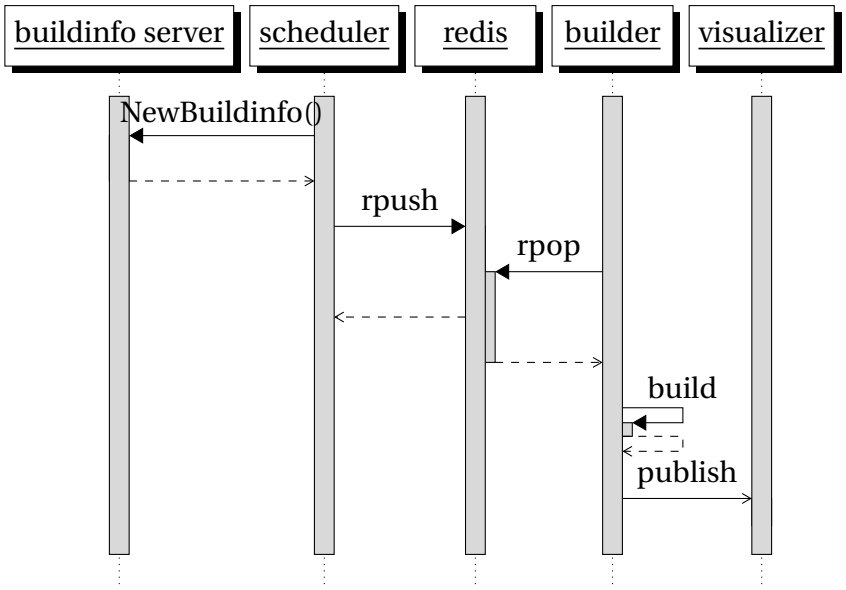


Figure 5.1: Rebuilder sequence diagram

The overlying architecture is displayed in 5.1 as a sequence diagram. It gives a quick overview of how the different parts interact with each other in sequence for one successful package rebuild. The scheduler queries the API of the buildinfo server for new BUILDINFO files since last time this was done. The scheduler pushes this file to a Redis, which is a very simple key value storage system. The builder queries the Redis server for new BUILDINFO submissions it is suppose to build. If the build is successful, it will produce a in-toto link metadata file, and a BUILDINFO file for this build. Both of these are submitted to the visualizer which publicizes in a public API which allows APT clients to query them.

5.1.1 buildinfo.debian.net

The goal is to rebuild packages released by Debian, but getting this information directly for a Debian package mirror can be tedious. What we instead do is relying on the buildinfo server

created by the Debian project to keep track of all published buildinfo files from built packages. This gives us a canonical view of all packages built by the Debian infrastructure.

To utilize this service we need to keep a track of all newly submitted files, however the current API does not support this. To get around this we submitted a code change so we would be able to get all files submitted after a given timestamp. This code change was not accepted in time, so the current rebuilder testing was done by setting up a copy of the server with the change included.

See Appendix A.3 on page 95.

5.1.2 scheduler

The scheduler is a small service which monitors the endpoint and schedules any new files found from the buildinfo server. Currently it pushes new package files to Redis, which is a very simple key value store, to help schedule the builders. This enables us to add an arbitrary number of builders. This is important for a few reasons. It helps scaling the system if its needed, and it also allows to have builders with different architectures to build packages.

Because of builder constraints the current scheduler does not add builds on other architectures then “amd64”.

5.1.3 builder

The builder consists of a service that queries Redis after new items on a timer. When new builds are dispatched, the build is done by utilizing the buildinfo files as provided by the Debian build server. The build are done with the tool “srebuild”.

“srebuild” is a Perl script used to build packages in a clean environment. With this environment the buildinfo is parsed and all missing dependencies are acquired to recreate the package. The source packages, which contains the source and the build files needed to build the package, is acquired from a mirror and the build is done. When the build is done, the results are signed with a cryptographic key, to verify that the build server produced the files, and then published to the visualizer.

5.1.4 visualizer

The visualizer is the component which displays the rebuilt packages in a web UI. The user is also able to fetch the buildinfo and link metadata files. The current implementation is a short

snippet of code backed by a SQLite database to aid in displaying the needed webpages. The implemented API as seen in 5.1 is simplistic and provides the needed features to let users verify builds.

The importance of the visualizer is that it is the component the APT package manager uses to fetch link metadata it utilize uses to verify packages before installation. Because of this we need to extend the component with a better architecture and implement a transparency log to make sure the submissions are not altered by a malicious rebuilder.

Endpoint	**Type**	**Parameters**	**Description**
/new_build	POST[1]	metadata, buildinfo	Submit a new build
/sources/<name>	GET		Gets the available builds for a package
/sources/<name>/<version>/buildinfo	GET		Gets the most recent BUILDINFO file for this version
/sources/<name>/<version>/metadata	GET		Gets the most recent in-toto link metadata for this version

[1] Behind authentication

Table 5.1: Old visualizer API

The Table 5.1 shows the current API implemented. It allows for submissions, and basic fetching of the available build submissions in the form of the BUILDINFO file and the link metadata file. This is the API that needs to be supported by the rewrite of the visualizer component to be compatible with the current rebuilder system.

5.2 Project development

We have now taken a look at the current implementation of the rebuilder system, and how it integrates with the current iteration of the visualizer. In the next session we will explain the development of the system for this thesis. It's structured in 4 iterations of the visualizer, and an integration with the existing APT transport written for the initial rebuilder system. We will in the first iteration tackle the problem of maintaining compatibility with the current system. The second iteration will be focusing on the implementation of the raw Merkle tree needed for transparency log. The third iteration will be the abstracted logic on top of the transparency log, which the APT transport will be utilizing when validating packages for the users.

5.2.1 First Iteration: Visualizer

Goals

The first iteration is largely focused on recreating the functionality of the current visualizer. The purpose of this component is to accept new rebuilds, the in-toto link metadata file and buildinfo file produced by the rebuilder. This needs to be displayed in a simple webpage and introduce no new features to remain compatible.

Thus the goals for this iteration are as follows;

- Maintain compatibility with the old visualizer
- Accept new build submissions
- Display packages with build submissions
- Display the submission files given a package name and version

Development

The initial task for this project was figuring out a structure that was flexible and made sense for the further development. The structure that was aimed for was to separate database models in its own directory, templates in its own directory and views in it's own. When this setup was done we could continue with the development of the visualizer.

In practice the visualizer only accepts two files, and displays an index of these files. There are no processing being done except to figure the given package name and version. The goal of this rewrite is to create a robust foundation where we can improve on the current design, and in later iterations build the needed data structures.

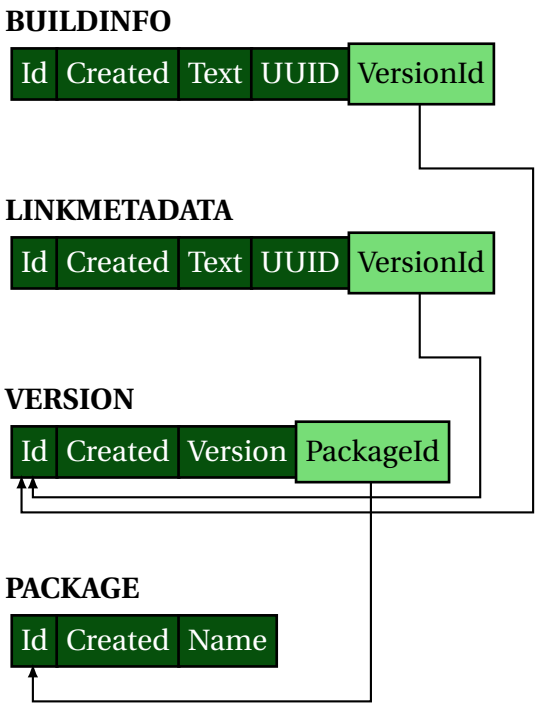


Figure 5.2: Database schema

The first step is to make sure the database format is correctly represented. The previous iteration had a strict dependency on SQLite, which is a very simple database stored in a single file. This works well for point of concept implementations and where the database does not grow exceedingly large.

In the rewrite we will be utilizing an ORM for Python, the SQLAlchemy library. This will allow us to define data models and instantiate them on top of different database engines. The database structure itself closely copies the one from the original implementation. The schema as displayed on Figure 5.2, page 36, represent the implemented model in the ORM.

The schema implements the model as follows; “package” can have multiple “versions”. The models that belong to “linkmetadata“ and “buildinfo“ is the tables containing the data itself. In the previous iteration these were stored as plain text files, which is a less portable way of dealing with the data. There can be multiple submissions for each version, so this relation is a one-to-many relationship where one “version” can have multiple submissions from rebuilders.

The “UUID” field found in the “linkmetadata” and “buildinfo” model is mostly a hack. The main issue was to find the pairs of submissions without over-complicating the database structure. One solution would be to create a new table to associate the submissions. This would enable us to properly group them later on and find the individual pairs. However, because of time constraints, and because the implementation of a new model would take some time, the addition of an unique “UUID” for each submission is an easier alternative that is simple to implement.

This allows us to group the submissions for the frontend later on.

It should be noted that the ORM handles the one-to-many and many-to-many relationships. They are not explicitly included in the modeled schema for on for the sake of brevity.

```
class Version(db.Model):
    __tablename__ = "version"
    id = db.Column(
            db.Integer(),
            index=True, unique=True,
            primary_key=True, autoincrement=True)

    created = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)
    version = db.Column(db.String(64), nullable=False)

    buildinfo = db.relationship("Buildinfo", back_populates="version")
    linkmetadata = db.relationship("LinkMetadata", back_populates="version")

    package_id = db.Column(db.Integer, db.ForeignKey("package.id"))
    package = db.relationship("Package", back_populates="version")

    def __repr__(self):
        return "<Version: {}>".format(self.version)
```

Listing 5: Sqlalchemy code for the Version model

The SQLAlchemy library enables us to define database models as native Python classes. As can be seen in Listing 5 we are able to represent the needed files, and relationships for the “version” model. In total the implementation consists of 4 classes which we are able to query, create and update. This lets us avoid having to implement the transition from the database format to the correct data representation in the project.

API and Frontend

To reimplement the API as can be seen in Table 5.1 on page 34, we need to create the needed routes for the web service. This is being done by Flask, which is a webserver framework for Python. The same framework was utilized in the original implementation and enables us to mainly substitute the code from the old database implementation, to the one utilizing the ORM. There are two parts to this, one part needs to return the plain-text elements for the APT transport, and one part needs to render HTML webpages for the user to browse.

```python
@app.route("/sources/<pkgname>")
def all_sources_pkg(pkgname):
    entries = (
        db.session.query(Package, Version, LinkMetadata, Buildinfo)
        .join(Version, Version.package_id == Package.id)
        .filter(Package.name == pkgname)
        .filter(LinkMetadata.version_id == Version.id)
        .filter(LinkMetadata.uuid == Buildinfo.uuid)
    ).all()
    return render_template("source.html", package=pkgname, entries=entries)
```

Listing 6: Python code for source.HTML

The listing seen in Listing 6 is a simple example of a route written in Python. It calls out to the needed models, and does an implicit join between them to get the needed relationships in place. The results of this is a webpage containing all the rebuilder submissions from the builder. Similar endpoints are written to provide rest of the functionality as described in Table 5.1. The input for this route is the name of the package and the version of the given package.

```
<!DOCTYPE html>
<html>
  <head><title> {{package}} </title></head>
  <body>
    <table>
      <tr>
        <th> Version </th>
        <th> Timestamp </th>
        <th> Buildinfo </th>
        <th> in-toto metadata </th>
      </tr>
      {% for entry in entries %}
      <tr>
        <td> {{package}}-{{entry[0].version}}</td>
        <td> {{entry[1].created }} </td>
        <td>
            <a href="/sources/{{package}}/{{entry[0].version}}/buildinfo">Link</a>
        </td>
        <td>
            <a href="/sources/{{package}}/{{entry[0].version}}/metadata">Link</a>
        </td>
      </tr>
      {% endfor %}
    </table>
  </body>
</html>
```

Listing 7: jinja2 template for source.HTML

To display these results, we are utilizing jinja2 templates that lets us compose HTML and Python code to generate webpages on the fly. The code as shown on Listing 7 is how we render the results of Listing 6 which contains all the submissions given a package and version.

REST API Considerations

The models we create rely heavily on relationships to properly store the information across several tables. Since we also back-reference across models, we end up in a peculiar situation. When we turn the models to a JSON structure, or attempt to debug them with the relationships enabled, they will recurse until the global stack limiter in Python is hit. This makes it somewhat hard to output the models in a good manner when debugging and developing the models.

```python
def recurse(func):
    @wraps(func)
    def wrapped(*args, **kwargs):
        if len(inspect.stack()) > 25:
            return None
        return func(*args, **kwargs)
    return wrapped
```

Listing 8: Python recurse limiter

```python
class Package(db.Model):
    __tablename__ = "package"

    @recurse
    def to_json(self):
        j = OrderedDict()
        j["id"] = self.id
        j["date"] = self.created
        j["name"] = self.name
        j["versions"] = list(map(lambda x: x.to_json(), self.version))
        return j
```

Listing 9: Python recurse limiter usage

For debugging and development purposes, there was a simple REST API developed to inspect the created objects when submitting new rebuilds. These would recurse indefinitely and crash the application. To counter this we wrote a simple hack to make sure the models would not indefinitely recurse. Since we are capable of inspecting the stack in Python, we wrote a simple wrapper to the functions prone to the recurse problem.

Listing 8 is a short and simple function which inspects the stack for every function it is wrapping. We are using the python syntax sugar for decorators. They are functions which wrap around another function, with an added syntax for the sake of clarity. The “recurse” decorator will be invoked before the function it is wrapping gets called. Here we simply check if the stack is larger then 25, in which case we stop calling the JSON deserializer and return. This is a slight hack, but it works to limit the depth of the call stack when outputting the raw models.

Listing 9 shows how we create the JSON objects. Since we want consistency, we utilize OrderedDict which is a dictionary reimplementation keeping insertion order of the items. This enables us to output the objects in a consistent manner.

The current visualizer doesn't utilize any sort of REST API, but we did write a few REST API endpoints for testing purposes and aid in debugging. This also laid down the ground work for the future REST API implementations. This approach worked wonderful in the next iterations of the project for debugging and API purposes.

Results

The result of this development is a mirror of the previous visualizer implementation. The requirement to achieve compatible interfaces is to be able to query the most recent buildinfo and link metadata file. The fact that it only is capable of querying the most recent is a design choice from the first revisions as it only stored the last submitted files. For compatibility reasons this is kept in the rewrite.

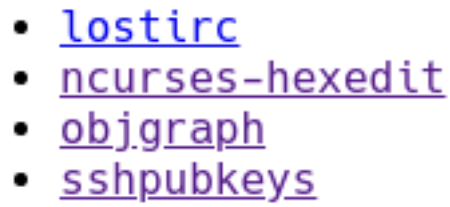


Figure 5.3: Overview of rebuild packages

Version	Timestamp	Buildinfo	in-toto metadata
sshpubkeys-3.1.0-1	2019-04-27 14:12:17.435077	Link	Link
sshpubkeys-3.1.0-1	2019-04-27 14:15:29.565981	Link	Link

Figure 5.4: Overview of the rebuild submissions

The website as shown on Figure 5.3 and Figure 5.4 displays the finished webserver for the initial rewrite of the visualizer. The resulting code and webpage maintains compatibility of the existing visualizer and enables us to continue implementing further improvements on the API.

5.2.2 Second Iteration: Merkle Tree

The second iteration focuses on implementing Merkle trees into the visualizer. These will create the foundation of the transparency log in the upcoming iterations for this project.

These will be implemented alongside of the reimplementation of the visualizer, and the current structure of the code. We will take a look at the challenges of implementing this correctly along with making sure the proofs are correctly implemented to support a transparency log.

Goals

The goals of this iteration is to make sure the Merkle tree operated properly. For this to happen there needs to be validated proofs and a proper tree generated when new items are added. We will also be writing a very simple output where we can visualize the generated tree as a node graph. This will aid in debugging the creation of the tree.

- Implement the needed database models for the datastructure
- New items should be appended to the tree
- The tree should not be rebuilt for each append
- Audit proofs should be implemented
- Consistency proofs should be implemented
- Visualize the tree graph

Development

The main challenge in this iteration is getting the underlying data model correct. What we are trying to achieve is to implement a tree structure where the hash of the leafs is hashed together to form a chain of checksums all the way up to the tree root. The following algorithm to achieve this is described below.

The development of this project will be done in a few stages. First we will take a look at the underlying database model needed to construct the trees. Next up is how to construct Merkle trees without rebuilding all interior nodes. We need to have a way to append new nodes by rehashing the least amount of interior hashes.

The next development challenge in this iteration is to make sure the proofs are working. Without appropriate proofs, mainly audit proofs and consistency proofs as described in Subsection 2.5.1. We will go through the development of these features and make sure they are working before moving on for the next iteration.

As noted in Subsection 3 on page 19, we are utilizing the pymerkle library written by Foteinos Mergoupis to aid in the creation of the transparency log proof algorithms [24]. It should be noted that this library does not provide a persistence storage outside of just storing the plain text tree as a JSON structure in a file. Our goal is to reuse portions of the library code, and adapt them to a tree structure backed by PostgreSQL to provide the transparency log features. This allows us to have a proper backend storage for the Merkle tree.

Database

Since we have implemented the visualizer with SQLAlchemy, the idea is to utilize the SQL backend to store the tree. This gives a few challenges on how to structure the model when it comes to relationships. We need to traverse the tree for the proofs developed later on, so the relationships bindings in the model should be correct and be easy to differentiate.

```
class Node(db.Model):
    __tablename__ = "node"
    id = db.Column(
        db.Integer(), index=True, unique=True, primary_key=True, autoincrement=True
    )
    leaf_index = db.Column(db.Integer(), default=0)
    type = db.Column(db.String(10), nullable=False)
    hash = db.Column(db.String(128))
    created = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)
    data = db.Column(JSONB)
    height = db.Column(db.Integer(), default=0)
    children_right_id = db.Column(db.Integer, db.ForeignKey("node.id"))
    children_left_id = db.Column(db.Integer, db.ForeignKey("node.id"))

    right = db.relationship("Node",
            foreign_keys='Node.children_right_id',
            uselist=False,
            backref=db.backref('children_right', remote_side=[id]))

    left = db.relationship("Node",
            foreign_keys='Node.children_left_id',
            uselist=False,
            backref=db.backref('children_left', remote_side=[id]))

    children_id = db.Column(db.Integer, db.ForeignKey("node.id"))
```

Listing 10: Node SQLAlchemy model

Listing 10 shows the model that was settled on. Each of the fields serve a purpose in the current model. “leaf_index” keeps track of what position as leaf node it was inserted as. This is important when we start constructing proofs as some of the lookups needs to know what leaf number 16 is. “type” denotes one of two things, “data” for leaf nodes or “level” for interior nodes. “data” is a leaf node that has the “data” field filled with JSON. “level” has no “data” field, but does have both “left” and “right” filled with the appropriate child node. “hash” contains the hash of the “data” JSON object.

The relationship modeled was a bit difficult to get correctly modeled in SQL. The intentions were to have one foreign key to bind the left and right relationships to, but this turned out to be difficult to implement. The solution was to have the “left” relationship bind to “children_left_id” and “right” bind to “children_right_id”. This solves the problem of having the relationship correctly modeled. To find the correct child node for the leaf and interior node, we need to check both left and right to find the correct node.

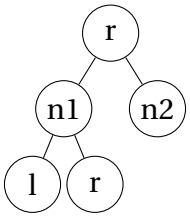


Figure 5.5: Example for relationships

In the example node relationship in Figure 5.5 we can demonstrate what the ORM described in Listing 10 would end up looking like. If we sit with the appropriate model for “N1” we have an interior node with a parent, left and right relationship. The parent node “r” where we can peek at the values “children_left” and “children_right” values to find the correct node. To find the “l” and “r” leaf nodes in this example we only have to check the “left” and “right” relationships respectively.

We are keeping everything in the same table with the same model. There is no inherent distinction between leafs, interior nodes and root nodes in this implementation when it comes to the model except for what is specified in the “types” direction. To find the root node in this implementation we only have to look for the last inserted “level” node.

Construct tree

We now have the database model we can build the tree with. Since this system is supposed to run over time, and might target thousands of nodes, we need to be able to append to the tree without rebuilding the entire tree. This is a tedious process and could very well take quite a

while if the number of nodes grows large enough.

The algorithm to build a tree, where all the interior nodes are hashed for every new insertion involves counting the number of nodes to be hashed. If this is an odd number, promote the last node in this list to the next step. Now we take two and two nodes and hash them together, left to right. This creates interior nodes we call "levels". When this is done, we start from the beginning, and the current set of interior nodes are the nodes we operate on.

This is a tedious process and does not scale very well over time. What we do need is an algorithm that only rehashes the changed interior nodes.

The algorithm for this involves figuring out all the loose subroots of the tree. This approach to append new nodes has been described in the original paper by Crosby and Wallach where they calculate frozen subroots down on the left side of the tree until the appropriate subroot is reached [15].

Our implementation is as follows; we take the count of all leaves in the tree and use them to calculate the largest complete subtree in the Merkle tree, and remove this from the count. What we are left with are all the nodes in surplus on the right hand side of the tree.

We can then walk down the tree on the left side, append the current node on the path, and then do the same calculation with the nodes in surplus. We then know the next complete subtree in the chain, remove the amount of nodes the complete subroot has from the count, append the node and walk down. We continue this until we reach 0 loose nodes, in which case we are at the node we are supposed to append the new node too. The chain of appended nodes upwards is the list of nodes we need to append with to rehash all of the interior nodes of the tree.

The resulting code takes the new node that is going to be appended, and hashes it in the with the most recent node, then walks up the chain. This only rehashes the interior nodes of the tree that needs to be rehashed. This lets us very easily reach a fast append-only Merkle tree implementation backed by an SQL database.

Cryptographic attack mitigation

When constructing hashes we need to consider the two sources of information we are essentially hashing. If the node is a leaf we need to hash the JSON data structure. This is done by taking the keys and values in the JSON structure and hashing them together in order. If the node is an interior node, this step is omitted and the hash consists of the hash of the left and right nodes as shown in Listing 11. The result of this is the appropriate hash for the node.

H\{NodeType, LeftNode, RightNode\}

Listing 11: Node hashing strategy

There are a few attacks one can do in practice on any data structure that uses cryptographic hashes. Since the same output always returns the same value, they are prone to what is called a “second preimage” attack. If the attacker knows the input, they can re append the values, and create the same hash in the tree. This can allow the construction of malicious data or open up for other venues of attack [30].

To mitigate this, a certificate transparency log appends a null byte value for leafs, and a 1 byte value for levels in the tree. This makes sure that we can’t replay previous data as there are always some values being appended. In our implementation we append “data” and “level” as the prefix in the hashed value. This serves the same purpose as the certificate transparency logs.

Graphviz

To validate that we are getting the correct association and relationships when appending new elements, one of the important things to get is a visualization of the currently constructed Merkle tree. To aid the discovery of problematic nodes, the first 10 letters of the hash is printed with the node to easily find the affected node and aid debugging.

To do this we loop through all of the nodes in the tree. We print out the left, and right parent of the node in a format that is compatible with the Graphviz tool to generate pictures [23]. The output found in Listing 12 enables us to produce the PDF picture in Figure 5.6 by supplying the output to “dot -Tpdf”.

It should be noted that when trees get sufficiently large, the graph itself is of less value when debugging. It is hard to get a good overview when the tree reaches around 60 nodes as the generated trees are spacious. Not enough time was invested trying to fiddle with the style got a proper display when the trees grow large. However, they did aid in early debugging when trees were sparse.

```
$ curl 127.0.0.1:5000/api/log/tree/graphviz
graph graphname {
labelloc="t";
label="Nodes: 4";
"9A565DAB05" -- "B439C9F4B0";
"9A565DAB05" -- "730212F389";
"B439C9F4B0" -- "47A2AB3D2E";
"B439C9F4B0" -- "648D1817A3";
"730212F389" -- "92EF30AD83";
"730212F389" -- "1E94F9FDE4";
}
```

Listing 12: Example graph of a generated tree

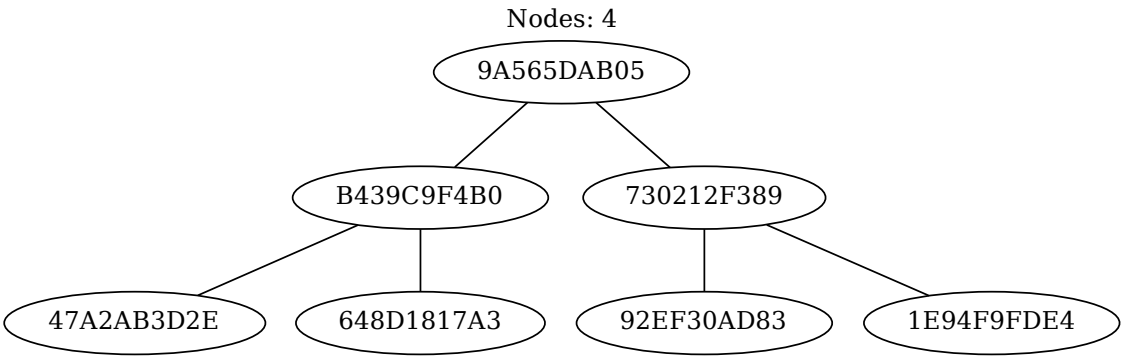


Figure 5.6: Graphviz visualization

Audit proof

Audits proofs are important as they prove whether or not a given leaf is part of the tree. They are used to validate that the data on the leaf is actually part of a Merkle tree. The implementation of this is fairly straight forward as we have implemented the database structure in a succinct manner.

The audit proof needs the id, or the hash of the node that should have the audit proof generated. When we have the node, we essentially traverse the children of the node. We then check on what side of the child the parent node is on. If it's on the left side, we fetch the right node on the next child. This node is appended to the list with the keyword "RIGHT". If it's on the right side, we

fetch the left node on the child. This node is appended to the list with the keyword “LEFT”. We then traverse to the next child and continue the path upwards until the root node is reached, which has no child node.

When all the nodes are appended to a list, with the correct side expressed we have a complete list of all the nodes needed. We can then append the current root node to the response and let users hash the path and recreate the current root node.

The appropriate JSON response for an example tree can be found in Listing 15. For the sake of being terse, some auxiliary fields have been omitted in this example. In this case, to recreate the root found in the top of the JSON output, we would just have to hash the “path” as given and append the hash on the correct side of the previous hash.

Consistency proof

Consistency proofs are very similar to audit proofs, however they carry another important property. They allow us to follow the path from a given known signed Merkle tree root to the current new one. They are important to make sure we are capable of tracing a path from the last known tree root we have, and to the current one.

To achieve this we need in practice two things; a known signed tree root and the number of leafs present at this tree root. The current algorithm does this in two steps. Procure the subroots needed to recreate the previous signed tree root, then use these to recreate a path to the current tree root.

The way to do this is to take the number of leaf nodes present in the input we are given. We can use this to figure out all the expected complete subtrees in the binary tree. These are represented as the expected heights of the given roots. We can then take the heights, and start from the first leaf, move up the tree until the root of the largest complete subtree is found. Then we know how many leafs we have to move forward until we can start to climb the next subtree.

Once this is done with all the found heights, we have all the needed roots to recreate the previous Merkle tree root. This can be checked by taking the previous tree root which is given as the input and do at the same hash concatenation as we do with the audit proof. If this value is as expected, we can continue recreating the path to the current tree root.

Here we take each of the previous subroots and move up the tree until we reach the current tree root. We end up with two paths. One consisting of the previous subroots, and one of the remaining interior nodes we need to complete the path to the Merkle tree root.

Results

The second iteration was all about finishing up the Merkle tree approach so we can construct a transparency log. The result has been a great deal of code to make sure the tree works, and the properties and paths are generated correctly. We have also made some debug utilities to make sure we have generated a well formed binary tree, and no nodes are generating poor relationships.

Endpoint	**Type**	**Parameters**	**Description**
/api/log/stats	GET		Statistics from the current tree
/api/log/graphviz	GET		Outputs the current tree in the dot format
/api/log/root	GET		Gets the latest tree root
/api/log/tree/append	POST	JSON object	Appends a JSON object to the tree
/api/log/tree/id/<id>	GET		Gets the given Node object from the database with '“id”
/api/log/tree/hash/<hash>	GET		Gets the given Node object from the database with “hash”
/api/log/tree/leaf/<id>	GET		Gets the given leaf from the database with matching “leaf_index”
/api/log/tree/validate/id/<id>	GET		Gets the audit proof for a leaf matching the given “id”
/api/log/tree/validate/hash/<hash>	GET		Gets the audit proof for a leaf matching the given “hash”
/api/log/tree/consistency	POST	ConsistencyQuery	Provides the consistency proof for the given query

Table 5.2: Second Iteration: Transparency log API

The Listing 5.2 shows the complete API that was designed to along with the code base. This API lets us add arbitrary JSON objects to the Merkle tree, and to inspect these nodes and verify them with an audit proof or consistency proof.

The API allows for inspection of the tree with graphviz, along with also inspecting individual interior nodes and leafs. It gives endpoints for the most important features, namely the audit proof and the consistency proof.

The goals stated in the beginning of this iteration have been met.

An appropriate database model has been constructed, and given us the needed relationships between nodes to form a tree, and effectively ports the needed algorithms to this tree structure. By lifting some code from the pymerkle library we are able to create append-only trees without having to rehash all interior nodes of the Merkle tree.

We are also capable of visualizing the created tree with graphviz for easier debugging when

dealing with the Merkle tree and detecting poor relationship building and find other issues in the data structure.

To test this we are going to take a look at creating Merkle trees, issue audit proofs, consistency proof and use the hashing scheme earlier to recreate root hashes. This will help us prove a well functioning transparency log can be achieved.

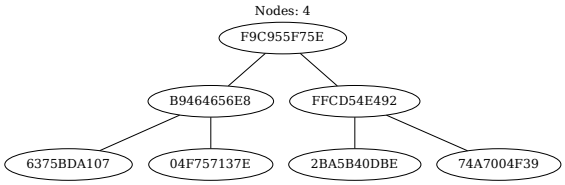
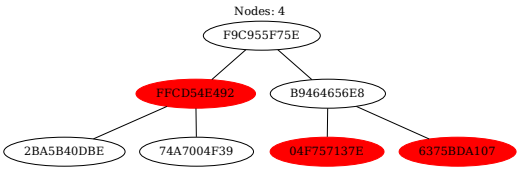
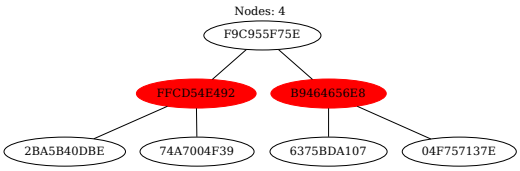


Figure 5.7: Graphviz visualization of tree

Figure 5.7 shows the constructed Merkle tree. The first level is the hash of all the leaf nodes which is currently some simple mock data in the JSON structure for testing purposes. There is a total of 3 interior nodes, where the top one is the Merkle tree root. Our goal is to test the appropriate proofs and make sure we are getting back the appropriate nodes. These will help us make sure we can reconstruct the root node in the tree.



(a) Validation proof nodes



(b) Consistency proof nodes

Figure 5.8: Proof nodes

Figure 5.8 shows the needed nodes for the different proofs. They have been artificially colored in red after being fetched from our graphviz API endpoint. The red color marks the needed nodes to construct the tree node for each appropriate proof.

It should be noted that while annotating the output graphs I discovered that the graphviz output as displayed in Subsection 5.2.2 is inherently unordered, at least for complete subtrees. Incomplete subtrees would always be output on the right hand side of the graph, but graphviz does not guarantee any ordering of the internal subtrees. That means the graph as shown in these examples does not completely represent the ordering of the internal tree, but this has no bearing on the result of the generating paths at all. It's mostly a minor inconvenience.

The proofs in Figure 5.8 are focused on leaf number two, in the displayed graph leaf number three. A bit hard to follow, but we will endure!

[(Side, NodeObject), (Side, NodeObject) ...]

Listing 13: Path structure

The way the path proofs are structured is a JSON object in a structure like shown on listing 13. We have tuples with two elements. The first specifies the side which the hash should be concatenated on, this can either be “LEFT” or “RIGHT”. The node object is the JSON structure we get from the Node object as defined in the ORM model.

```
1  def validate_chain(root, chain):
2      if not chain:
3          return True
4      chain = chain[:]
5      s = chain.pop(0)[1]["hash"]
6      for node in chain:
7          h = hashlib.sha512()
8          if node[0] == "LEFT":
9              h.update(("level"+node[1]["hash"]+s).encode('utf-8'))
10         if node[0] == "RIGHT":
11             h.update(("level"+s+node[1]["hash"]).encode('utf-8'))
12         s = h.hexdigest()
13     return root["hash"] == s
```

Listing 14: Code for validating path

To concatenate so we achieve the present root node, we need a function to hash each of the tuples on the correct side. The Listing 14 displays this algorithm used in this project. There are a few important points.

The lines 2 and 3 deal with the possible issue of having an empty list hashed. This is more of a short circuit in the case we are dealing with empty paths. Line 4 explicitly deals with the fact that Python is designed to be pass-by-reference. Any modifications on the path when we pop off items to create the hash will affect any code holding the reference. We thus copy the list to make sure no references are modified. The rest of the code is all about concatenating the strings on the correct side of the nodes to create the hash.

```
$ curl 127.0.0.1:5000/api/log/tree/validate/id/2
{"root": {
    "id": 8,
    "type": "level",
    "hash": "f9c955f75e...",
    "right": "ffcd54e492...",
    "left": "b9464656e8..."},
  "path": [
    ["RIGHT", {
        "id": 2,
        "type": "data",
        "hash": "04f757137e...",
        "parent": "b9464656e8...",
        "data": {"data": "Datablock-1", "name": "Name-1"}}],
    ["LEFT", {
        "id": 1,
        "type": "data",
        "hash": "6375bda107...",
        "signature": "8acfc7f730...",
        "parent": "b9464656e8...",
        "data": {"data": "Datablock-0", "name": "Name-0"}}],
    ["RIGHT", {
        "id": 7,
        "type": "level",
        "hash": "ffcd54e492...",
        "right": "74a7004f39...",
        "left": "2ba5b40dbe...",
        "parent": "f9c955f75e..."}]],
  "validation": true}
```

Listing 15: JSON for audit proof

Listing 15 is the audit proof we need for leaf number 2 when calling the API. It should be noted that we do validate the path on the server side and return the result in the “validation” field, but clients shouldn’t trust this value blindly. This do make for easy debugging when we are running this locally. We are also abbreviating some of the structure and hashes for the sake of keeping things terse.

The Figure 5.8a on page 50 is the audit tree with the appropriately highlighted nodes, and the same nodes are found by the audit endpoint and a path to the root node is correctly generated. This can be verified running the given structure with the previously mentioned hash function.

In our case we can also trust the initial validation done by the server, as it uses the same function to hash and compare the root node. However our client implementation should not rely on this.

```
$ curl 127.0.0.1:5000/api/log/tree/consistency/2
{"root": {
    "id": 8,
    "type": "level",
    "hash": "f9c955f75e...",
    "right": "ffcd54e492...",
    "left": "b9464656e8..."},
  "leaf nodes": 4,
  "inclusion": true,
  "consistency": true,
  "path": [
    ["RIGHT", {
        "id": 7,
        "type": "level",
        "hash": "ffcd54e492...",
        "right": "74a7004f39...",
        "left": "2ba5b40dbe...",
        "parent": "f9c955f75e..."}],
    ["LEFT", {
        "id": 3,
        "type": "level",
        "hash": "b9464656e8...",
        "right": "04f757137e...",
        "left": "6375bda107...",
        "parent": "f9c955f75e..." }]]}
```

Listing 16: JSON for consistency proof

Listing 16 displays the consistency proof. It is considerably smaller then the audit proof as we are only dealing with the subroots. Since we are dealing with a rather small tree, it is not the best situation to show off a consistency proof, as one subroot of the two is a previous Merkle tree root. Thus we are only getting two in return we need to hash together.

However, as shown on Figure 5.8b on page 50 we are returned the correct tree roots for the proof and thus we have implemented the needed proofs.

In Section 6 on page 71 we are going to investigate and test the transparency log implementation further.

5.2.3 Third Iteration: Tree root signing

One essential part of Merkle trees as implemented by Laurie, Langley, and Kaster in certificate transparency logs is the ability to have tree-roots signed [38]. This lets us verify with public key cryptography that the tree-root was created by the given transparency log. In this chapter we will go through how this is implemented.

Goals

The goals of this iteration is to create the needed code to create and verify signatures on tree nodes.

- Initialize signing keys
- Sign every tree root created by the log
- Verify the signature

Development

For the development of this feature we are utilizing the securesystemslib from New York University's' secure systems lab [47]. It's a library with a collection of easy-to-use primitives to deal with encryption, decryption and signature verification on data for Python. Utilizing this library makes it trivial to implement the needed signature functionality in a short and concise manner.

```
PASSWORD = "123"
KEY_NAME = "ed25519_key"

def init_keys():
    if not os.path.isfile(KEY_NAME):
        generate_and_write_ed25519_keypair(KEY_NAME, password=PASSWORD)

def get_private_key():
    init_keys()
    return import_ed25519_privatekey_from_file(KEY_NAME, password=PASSWORD)

def get_public_key():
    init_keys()
    return import_ed25519_publickey_from_file(KEY_NAME+'.pub')

def sign_data(data):
    return create_signature(get_private_key(), data)

def verify_data(signature, data):
    return verify_signature(get_private_key(), signature, data)
```

Listing 17: Glue code for “securesystemslib”

The implementation itself consists of a file for the needed code to ease the key creation for the rebuilder. For the sake of ease, we don’t consider problems such as password strength for the signing key as the main focus is on the transparency log implementation. In this implementation the password is hard coded to “123”, which is admittedly a poor password. For the cryptography, elliptic curves are used instead of the traditional RSA algorithm. The reason is that elliptic curves needs less bits to produce equally strong keys. We then end up with faster and smaller signatures.

These auxiliary functions makes it trivial to further implement the tree root signing in the model.

```
@@ -21,10 +22,10 @@ class Node(db.Model):
     leaf_index = db.Column(db.Integer(), default=0)
     type = db.Column(db.String(10), nullable=False)
     hash = db.Column(db.String(128))
+    signature = db.Column(db.String(128))
     created = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)
     data = db.Column(JSONB)
     height = db.Column(db.Integer(), default=0)
```

Listing 18: Additions to the Node model

In Listing 18 we can see the additions to the "Node" model from Listing 10 page 43. This enables us to store the signature string in the database model for future use when creating tree roots.

```
@@ -166,7 +168,9 @@ def append(data):
     for node in reversed(subtrees):
         new_parent = create_level_node(node, new_node)
         new_node = new_parent
+    signature = sign_data(new_node.hash)
+    new_node.signature = signature["sig"]
     db.session.commit()
     return ret
```

Listing 19: Additions to the append function

Listing 19 shows the additions made to the "append" function to support signed tree roots. Since the last "new_node" is the new tree root, we only sign the hash of this node. This is enough to implement tree root signing in the current implementation of the visualizer.

Results

The results of this iteration is two new endpoints and some new code to help with signing tree roots in the transparency log. The resulting API endpoints can be found in Table 5.3 which shows the newly created endpoints.

Endpoint	**Type**	**Parameters**	**Description**
/api/crypto/key	GET		Outputs the key object used by the library
/api/crypto/validate	POST	Tree root hash and signature	Validates the tree root to the public key

Table 5.3: Third Iteration: Crypto API

The usage of the endpoints is fairly straight forward. We can call the generated signing key with "/api/crypto/key" as shown in Listing 20. Which enables us to verify the signature on the client side instead of on the server side if we need to do that.

```
$ curl 127.0.0.1:5000/api/key
{
  "keytype": "ed25519",
  "scheme": "ed25519",
  "keyid": "25a16bb3a3...",
  "keyid_hash_algorithms": [
    "sha256",
    "sha512"
  ],
  "keyval": {
    "public": "7623ba359c..."
  }
}
```

Listing 20: Display generated public key

To verify a tree root on the server side, we can provide the output from "/api/log/tree/root" from Table 5.2, and forward this to "/api/crypto/verify". In the example from Listing 21, we have provided a minimal tree root JSON object to display a successful verification of a tree root on the server side.

```
$ curl --header "Content-Type: application/json" \
  --request POST \
  --data '{"hash": "21e0b13a6f...", \
           "signature": "832510c0188..."}' \
  127.0.0.1:5000/api/crypto/verify
{
  "status": "ok",
  "verified": true
}
```

Listing 21: Test of the verify endpoint

5.2.4 Fourth Iteration: Transparency log overlay

In this section we are going to create an abstraction over the values we insert into the Merkle tree. So far we have only been using testing values to test the correctness of the tree. But we are after all going to record the submissions from the rebuilder and append them to the log. We also need to expose a new API to the APT transport which we need to integrate with at a later point.

We want to have some enforced properties to see how we can implement abstractions on the values. We can't remove entries once they are on the transparency log, but we might want to retain the property of revoking package submissions. This could be because of build environment compromises, or security advisories from the parent distribution.

Goals

The goals for this iteration are as follows;

- We should have well defined entry formats
- We should be able to find package submissions
- We should be able to revoke package submissions

Development

First we need to figure out what data we are dealing with. Let us start off by defining one item appended on the tree as an "Entry". An entry should always contain two key pieces of information, namely package name and version. This is important to retrieve them at a later point so we are capable of doing the package verification with the APT package manager.

Once we have established this we need to figure out what sort of entries we are dealing with. We are going to deal with two actions, accepting submissions from the rebuilder, which we will call inclusions, and revocations of said submissions.

```
inclusion = {"type": "inclusion",
             "package": pkg_name,
             "version": version,
             "linkmetadata": metadata.read().decode("utf-8"),
             "buildinfo": buildinfo.read().decode("utf-8")}

revoke = {"type": "revoke",
          "package": pkg_name,
          "version": version,
          "hash": hash,
          "reason": reason}
```

Listing 22: Entry definitions

Listing 22 displays Python code to illustrate the two entries we are going to be dealing with on this transparency log. We have the “inclusion” entry which lists the package name and version. This is a common field assumed for all entries, along with the appropriate files that contain the buildinfo, which contains the record for the build environment.

We also have linkmetadata, which is the in-toto metadata needed for the client verification later on. This denotes that the package and the files has been rebuilt and is a submission.

The other entry we will be dealing with is a “revoke” entry. This denotes an inclusion which has been marked as bad. We will be using these entries to mask inclusion entries on the client side. This contains two fields. The “hash” field denotes the hash of the Merkle tree object containing an inclusion entry. The next field is “reason” which can contain an optional message as to why the given inclusion has been revoked.

To implement support for this we are going to take the submission endpoint from the first iteration on Subsection 5.2.1 on page 35.

Extending the previous “/new_build” endpoint, where we are creating the needed “Package” and “Version” models for database insertion, we now also append the relevant information to the transparency log. This is fairly straightforward as we only need to construct the “inclusion” dictionary and insert it as JSON into the database.

```
@app.route("/api/rebuilder/fetch/<name>/<version>")
@json_response
def rebuilder_fetch(name, version):
    records = (db.session.query(Node)
             .filter(Node.data.op('->>')('package') == name)
             .filter(Node.data.op('->>')('version') == version)
             .order_by(Node.created.desc())
            ).all()
    return [[rec.to_json(), audit_proof(rec)] for rec in records]
```

Listing 23: Fetch entries

We also need to fetch all entries on the database given a package name and version. PostgreSQL has dedicated JSON support in their database, which our model defined on Listing 10 on page 43 has implemented for the data field. What we can now do is use native PostgreSQL filters on the JSON as seen on Listing 23. This allows us to easily filter through all entries and return them easily for the REST API.

For each of the entries a corresponding audit proof needs to be fetched. To lessen the burden on the client, we append this proof to the response. This allows us to not having to implement the audit proof fetching and validation

When it comes to having a “revoke” entry, we need to have a few considerations in place. As input we need the hash of a node on the Merkle tree with a “inclusion” entry. Currently we don’t want to publish revocations of any objects, as further entries could be implemented in the future. What we do is that we look up the hash, get the object and verify the type of the entry before submitting the revoke entry on the object.

Results

The results of this iteration is the last and final API. This iteration completes the work on the rewrite of the visualizer and provides us with a new API that will be used for the upcoming APT integration.

The API as listed in 5.4 gives us the ability to submit new rebuilds, and at a later point revoke them if needed. All of this is stored on an abstracted API where we don’t have to deal with the transparency log directly.

Endpoint	Type	Parameters	Description
/api/rebuilder/submit	POST[1]	metadata, buildinfo	Implements something heyho lets go
/api/rebuilder/revoke	POST[1]	leaf hash, reason, signature	Revokes the leaf with a reason
/api/rebuilder/fetch/<name>/<version>	GET		Gets the entries for the given package

[1] Behind authentication

Table 5.4: Fourth Iteration: Overlay API

```
$ curl -F "metadata=@metadata" -F "buildinfo=@buildinfo" \
  127.0.0.1:5000/api/rebuilder/submit
{"status": "ok",
  "node": {
    "id": 1,
    "hash": "9432b4df00...",
   ...}}

$ curl 127.0.0.1:5000/api/log/tree/leaf/index/1
{"id": 1,
  "hash": "9432b4df00...",
  ...}
```

Listing 24: Example of rebuild submission

Listing 24 shows how we can submit a rebuild, consisting of the buildinfo file and the linkmetadata file, to the submit endpoint. This will create the appropriate models, and append it to the transparency log as shown when we call it with the appropriate index.

```
$ curl -XPOST --header "Content-Type: application/json" \
  --data '{"hash":"9432b4df00...","reason":"testing"}' \
  127.0.0.1:5000/api/rebuilder/revoke
{"status": "ok",
  "node": {
    "id": 2,
    "type": "data",
    "hash": "d22c2446cb...",
    "parent": "e5a6599214...",
    "data": {
      "hash": "9432b4df00...",
      "type": "revoke",
      "reason": "testing",
      "package": "lostirc",
      "version": "0.4.6-4.2"
    }}}
```

Listing 25: Example of rebuild revoke

Listing 25 enables us to revoke the inclusion on the tree and also allows us to state the reason for the revocation of the inclusion. This evidently works fine and allows us to fetch all of the entries on the test package. We are currently using a previously rebuilt package with a corresponding linkmetadata file. The package is called “lostirc” and the version is “0.4.6-4.2”

```
$ curl 127.0.0.1:5000/api/rebuilder/fetch/lostirc/0.4.6-4.2
[{"id": 2,
   "type": "data",
   "hash": "d22c2446cb...",
   "data": {
     "hash": "9432b4df00...",
     "type": "revoke",
     "reason": "testing",
     "package": "lostirc",
     "version": "0.4.6-4.2"}},
  {"id": 1,
   "type": "data",
   "hash": "9432b4df00...",
   "data": {
     "type": "inclusion",
     "package": "lostirc",
     "version": "0.4.6-4.2"
     ...}}]
```

Listing 26: Example of fetching entries

Listing 26 shows how entries are gotten in descending order of creation. First the revocation entry, then the inclusion entry. This allows us to read each entry subsequently in order, and act on the next elements according to previous entries. We can also see how we are capable of fetching multiple inclusion proofs for the same package as we are only dealing with the package name and the package version.

This API is flexible and can be extended to include multiple entries in the future. We will now attempt to integrate them into the APT package handler.

5.2.5 APT Transport integration

So far in this thesis we have implemented the server side of this software. Now comes the client side which will do the package verification towards the visualizer. The current rebuilder system as implemented before this thesis, there was an integration with the APT package manager for Debian written for this. The purpose of this integration is to have a process which let APT download the appropriate package, but in a way where we can do additional verification.

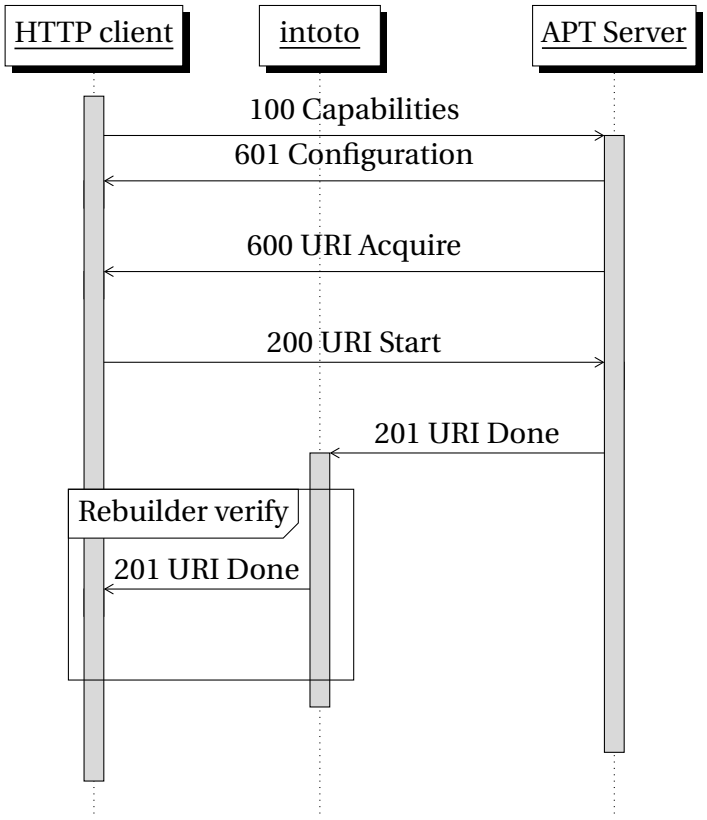


Figure 5.9: in-toto sequence diagram

The sequence diagram in Figure 5.9 shows the idea behind this. When APT communicated with a package repository it will use a binary for the communication between the client and the server. What we have is an "in-toto" client which can be used as a drop-in replacement of the "http" one. We only forward between the client and the APT server until we see that a package has been acquired, this is recognized when the "201 URI Done" message is passed. This implies we have downloaded the package locally. We man-in-the-middle this message and begin our verification process.

This process involves having configured one or more rebuilders we should be communicating with in a configuration file. We will then request each of the rebuilders on this list for the linkmetadata file. The verification step is implemented with "in-toto". This involves a layout from in-toto which describes the expected keys and the threshold that needs to be met for the process to succeed. If the validation with the linkmetadata is approved, which means the downloaded package has the same checksum as the signed layout says it has, we will forward the "201 URI Done" message to the client APT and the package will be installed.

The development in this section is to move the current implementation away from the old API, which is a very simple file listing, to the transparency log implementation.

Goals

For this integration we need to move the old API interactions to the new one. To do this we need to implement a few things.

- We need to store rebuilder tree roots
- We need to verify any signatures
- We need to verify the consistency proof
- Parse available entries
- We need to verify the audit proof of entries
- Validate with available metadata

An additional goal of the API is to validate the output without need to trust any validations the rebuilder does.

Development

To validate the downloaded packages against available linkmetadata, we need to implement and support a few states. We can't just ask for the given entries, we also need to validate the state of the Merkle tree with the consistency proof from the Merkle tree. All of this should be done locally.

The steps needed are

1. Have previous known Merkle tree state
2. Fetch consistency proof and validate this proof
3. Store new tree root in place of old tree root
4. Fetch available entries on the given package
5. Fetch audit proof for each entry and validate this proof
6. Validate package -with any valid inclusion entries

It should be noted that any validation step on tree nodes need to attempt to recreate the hash appropriately. We also need to validate any signatures on Merkle tree roots. Thus our additions to the "in-toto" transport amounts to 5 functions.

- compare_hash
- validate_chain
- verify_consistency
- store_new_tree_root
- fetch_rebuild

“compare_hash” will be our function making sure the node hash is correct. It will take the parents and data directive as appropriate and calculate the hash as a boolean value. “validate_chain” will utilize the “compare_hash” function and make sure the given path for audit proofs and consistency proofs are correct. “verify_consistency” will request the consistency proof and build on top the previous functions. It will take the rebuilder, leaf count and previous root and make sure the proof with the current tree root is signed along with the path validating.

Once this is done we can store the new tree root locally with “store_new_tree_root”. This leaves us with fetching the current entries of the given package and version which will be done with “fetch_rebuild”. It will loop over any given entry, and mask any hashes where we have a “revoke” entry. We will also make sure the consistency proof of the given entry is correct.

```
@@ -563,21 +649,21 @@ def _intoto_verify(message_data):
       .format(len(global_info["config"]["Rebuilders"])))
   # Download link files to verification directory
   for rebuilder in global_info["config"]["Rebuilders"]:
-    # Accept rebuilders with and without trailing slash
-    endpoint = "{rebuilder}/sources/{name}/{version}/metadata".format(
-        rebuilder=rebuilder.rstrip("/"), name=pkg_name,
-        version=pkg_version_release)
-
-    logger.info("Request in-toto metadata from {}".format(endpoint))

     try:
-      # Fetch metadata
-      response = requests.get(endpoint)
-      if not response.status_code == 200:
-        raise Exception("server response: {}".format(response.status_code))
+      proof = verify_consistency(rebuilder)
+      if not proof:
+        raise Exception("Can't verify consistency on rebuilder {}".format(rebuilder))
+
+      store_new_tree_root(rebuilder, proof)
+
+      rebuild = fetch_rebuild(rebuilder.rstrip("/"), pkg_name, pkg_version_release)
+      if not rebuild:
+        raise Exception("No available rebuild from {}".format(rebuilder))
+
+      link_json = json.loads(rebuild["data"]["linkmetadata"])

-      # Decode json
-      link_json = response.json()
+      logger.info('{}'.format(link_json))
```

Listing 27: Difference in “intoto.py” verification step

Listing 27 shows the difference in code for the verification step in the transport after the initial implementation. This implementation fetches from the new rebuilder API instead of the old one with the least amount of intrusive code to the current project.

```
{
    "http://127.0.0.1:5000": {
        "hash": "78f6bbe2f6...",
        "signature": "b8d0925acd...",
        "count": 15
    }
}
```

Listing 28: File format for storing tree roots

We have also included a new option in the configuration file for the APT transport. We need a way to store previous tree roots from multiple rebuilders, so settled on a fairly simple JSON format which meets our needs fairly well. Listing 28 shows the format which is a simple key value format where we store the three important pieces of information. The hash of a tree root, the signature of the tree root and the number of leafs present when this tree root was the current version. We write this file as appropriate when new tree roots are gathered.

The links to the source code of this can be found in Appendix A on page 95.

APT Transport Testing

For the testing part we are going to reuse the test setup written for the original APT transport and make sure it has everything it needs to test towards the new API.

The current test setup heavily relies on mocking three parts of this setup. The APT client, the APT server and the rebuilder. All of these are mocked in an attempt to only test the “intoto” transport. To test our new API we will remove the rebuild mocking which the test setup utilizes. This enables us to test the new rebuilder directly instead of utilizing a mock of the API.

First we need to modify the in-toto layout used for testing. The in-toto layout is used for making sure thresholds of available, and good, linkmetadata exists for the given package. The current test setup requires two good metadata files to exist, this is because they are utilizing the test setup with two rebuilders to make sure things work appropriately. In our case we only really care about the “intoto” transport communicating correctly with the new API. We modify the linkmetadata for the tests to only account for one valid linkmetadata file, then resign it with the correct key used in the setup with the “in-toto-sign” utility.

The next challenge is to make sure we are initializing the consistency proof tree root correctly. To do this we make sure we are dealing with a reproducible setup so the seeded Merkle tree root doesn’t change between tests. We do this by first initializing a Merkle tree with 10 elements.

Then we pick one of these roots which we will use to start off on our testing, in our case we went with the Merkle tree root present when there was four leaf nodes for no particular reason. We make sure we store this file, and copy it to the right location, along with making sure the mock APT client has the configuration set to look for this file.

The resulting test suite thus passes and we have managed to integrate the APT transport with the new API. Appendix B on page 97 goes through the test setup of the APT transport and the resulting test suite run. This helps make sure the results are reproducible.

Summary

In this section we have explained how the development of the new visualizer was done. What we got is a new visualizer where we have support for transparency logs and an extended API and code base to help further development in the future.

We have also done the needed integration between the rebuilder transparency log and verified it works.

Chapter 6

Evaluation

In this section we will be evaluating the project. We will be utilizing the outline described in Chapter 4 to help evaluate the final research artifacts we are left with.

So far we have had four iteration on this project. First iteration was to reimplement the API and functionality from the first visualizer, the second iteration implemented support for Merkle trees, the third iteration implemented the needed signing layer on the tree, and the fourth and final iteration that implements the transparency log overlay.

The APIs we created during the different iterations of this project is what we will be taking a look at first. For this evaluation the attributes for modular API design as outlined by Iyer and Wyner is going to be used [34]. We will then be doing a technical evaluation of the transparency log implementation by stress testing the data storage and comparing it to the expected amount of data from published Debian packages. The APT integration is going to be evaluated last so we can consider some of the possible security guarantees.

Appendix C on page 99 is going to contain any auxiliary details and information for this evaluation.

6.1 API Evaluation

The APIs we are going to evaluate are the transparency log API from Table 5.2 on page 49, the crypto API from Table 5.3 on page 56 and the transparency overlay API from Table 5.4 on page 61.

We are not going to consider 5.1 on page 34 as it predates this thesis and is mainly reimplemented for the sake of compatibility with the current rebuilder setup.

Functionality

This attribute centers around the need to make functionality offered intuitive. The developed APIs centers on the concept of standardized REST API concepts and should be easy for anyone familiar to utilize. The API also covers all the requirements to interact with the transparency log provided.

Hierarchy

When designing complex API there is a chance internal implementation detail leaks to the user of the API. In our current implementation internal details are exposed as we have incrementally developed our API. We are capable of adding pure JSON objects to our log, as well as adhere to the entry definitions for the overlay.

Separation of concerns

Separation of concerns in this scenario is handled well. We have split functionality across multiple APIs with clearly labeled names. The tree API does not implement functionality in the Merkle tree API. However, when utilizing the API one has to use APIs from both the transparency log API and the transparency overlay API.

Interopability

Interopability is how easy it is to interact with this API. Complicated APIs might be less useful as the tools available might be sparse. The way to interact with this API is through REST API calls. This is widely used and a lot of tools are available to interact with APIs designed in this fashion.

Reusability

The API is not designed to be inherently reusable for other usages than the one it is supposed to cover. We can however use the raw Merkle tree without utilizing the more abstracted API calls in this implementation.

6.2 Transparency log testing

To test the Merkle tree implementation we need to first consider the amount of data this system would realistically receive. We have implemented this tree on top of a SQL database, so performance impacts should be checked to see how large a tree can grow before any noticeable impacts can be found.

Debian builds and releases packages into a testing repository on a daily basis, this produces the BUILDINFO files we will be using to recreate packages in the rebuilder. This can be retrieved from a centralized server with an API. However, since the API was tedious to work with there has been put up a file server with all the BUILDINFO submissions. This allows us to grab the files produced for each day of the year.

The file server contains builds information from 2016 until 2019 and totals an archive of around 130 GB. As the Merkle tree grows for every insertion and hashes interior nodes along with signing the root node, we need to limit the data for practical reasons. We are going to only consider the build submissions from 1st of January 2019, until 19th of May 2019 (see Appendix C.2 on page 100).

Date	**Count**
2019-01	41566
2019-02	29750
2019-03	18354
2019-04	10061
2019-05	5288
Total	105019

Table 6.1: Debian package builds from 1st of January until 19th of May

Table 6.1 shows the number of builds done for each month. There is a slight spike in January with a subsequent falloff through the spring. The total submissions is around 105000. The goal of the testing is to see how well the current implementation can handle around 105000 elements on the tree. What is the performance, and is it acceptable for implementation in the real-world in its current state?

Testing setup

One of the goals as outlined in the introduction is the ability to reproduce the results of the thesis. For this we need to have a way to easily create the components needed for the testing. For the transparency log we need two components, one database service running PostgreSQL,

and one service to run the visualizer component. To make sure this setup is easy for future researcher to set up, we utilize a technology called containers. Containers are light weight process separations on the host operating system. The commonly used tool for creating containers is called Docker, as described in Section 3.

Docker allows us to define the components in a way that lets us easily build and tear down the systems without having to manually setup the services themselves. The testing is done with this setup to make sure reproducible results can be achieved. We also provide a "Makefile" which is a standardized format for executing build, testing and installation scripts for software. This allows us to provide a generic interface where one can run the test setup in the future.

The test data we are using is small key value pairs with an incremented number. The reason for this is because creating and faking the appropriate data would require a fair bit of parsing of the BUILDINFO files as link metadata is not included in the file server.

See Appendix C.1 on page 99 for the testing instructions.

Testing results

For the testing of the tree, there was around 113000 elements appended to the transparency log. The intentions is to test what would happen with the data storage when approaching an expected amount of build submissions on the log. As the transparency log is going to be used by the APT client to find the metadata needed to verify packages we need to see if the log interaction is going to slow down the process.

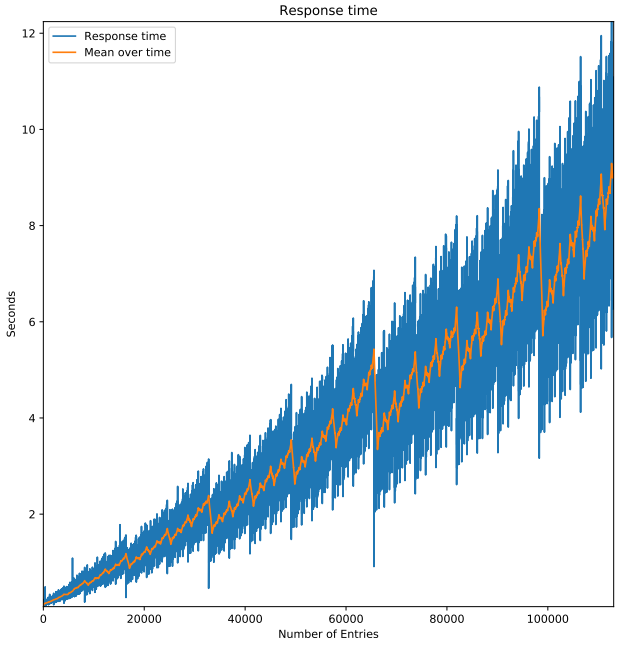


Figure 6.1: Response time on entry inclusion

Figure 6.1 shows the growth in response time when appending new entries to the log. The log has a blue line, which is the response time, and a orange line which is the mean response time as we move through the graph. The minimum response time was below 0.1 seconds, and the maximum response time observed was 12 seconds. The testing itself lasted around 120 hours, or around 5 days. The size of the table in complete ended at around 243 MB. Consistency and audit proofs also spend approximately the same time for each query.

12 second is not by any means a good metric in this case. We need to do multiple consistency proofs and audit proofs for each package we are going to install. This quickly adds up in seconds and slows down the package installation process by a non trivial amount of time. The main reason after some investigation is that relationship building is taking a fair amount of time.

PostgreSQL is not well optimized for joins and relations when it's done on the same table. This quickly adds up seconds when we do several queries to get the object models from the ORM in place.

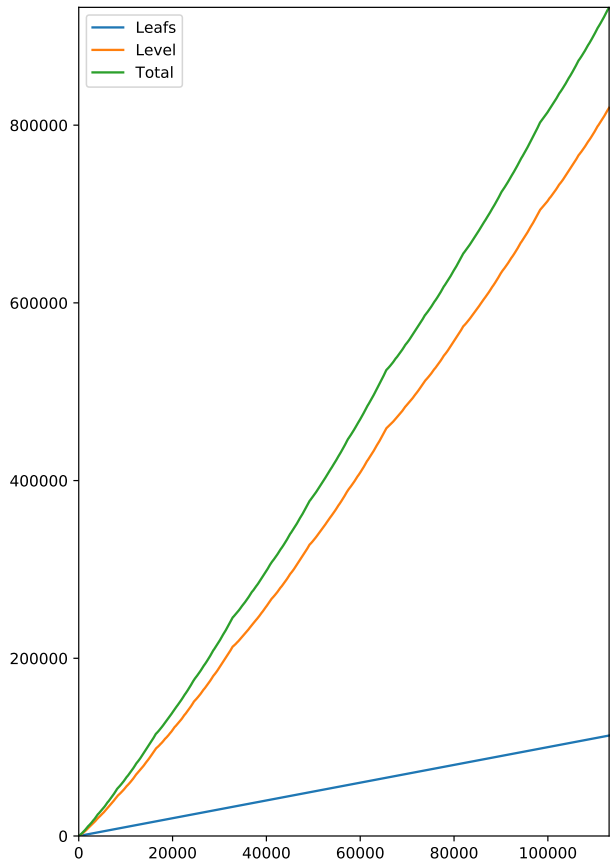


Figure 6.2: Number of nodes over time

Figure 6.2 shows the number of leafs and interior nodes that get created over time. For 113054 leafs a total number of 819533 interior nodes get created. This amounts to a complete 932587 objects on the database table. This suggests a naive implementation in the append function we implemented in Section 5.2.2 on page 42 In a proper implementation the number of interior nodes should mirror the number of leafs. This is the issue causing the poor performance of the tree.

However, the proofs works and has been correctly implemented as proven when we implemented it into the APT transport in Section 5.2.4 on page 60. The poor performance does not affect the correctness of the tree.

6.3 Summary

In this chapter we have taken a look at the implementation of the new visualizer component for the rebuilder system. We have assessed how well the implementation written in this thesis can

handle real-world data. The data we compared to was the number of package builds Debian have done over a 5 months period.

The assessment shows that the current implementation does not scale very well considering the amount of data it would be required to store. There needs to be more research into how this implementation can be improved.

Chapter 7

Discussion

In this chapter we will be discussing and reflecting upon the research methods used in this project and answer the research questions at hand.

7.1 Design Science Research

This research has been based on the Design Science research methodology. We have identified a problem and attempted to solve this problem by following the guidelines set forth by Hevner et al.; design as an artifact, problem relevance, design evaluation, research contributions, research rigor, design as a search process and communication of research [28].

Design as an artifact means that the product of this research should be something tangible in the form of a construct, a model, a method or an instantiation. This research has produced an instantiation in the form of a rewritten visualizer with a transparency log implementation.

The problem relevance implies that the artifact produced by this research should be relevant and solve an important problem. This guideline has been satisfied as we have attempted to combine two previously separate technologies; the rebuilder system with a transparency log.

Design evaluation means that we need to go through an evaluation of the created artifact. This was done through the Chapter 6 on page 71.

The research contribution guideline means that the research must provide clear and verifiable contribution in the specific areas the artifact is designed for. The artifact in this thesis contributed directly to securing the supply chain for Debian packages and can hopefully lead to further research in this area. The artifact was designed with clear means to reproduce the environments and tests. This thesis allows a clear verifiable and reproducible path to the results of the contribution.

Research rigor means that the construction of the artifact and the research has gone through rigorous methods in both construction and the evaluation of the artifact. We have done a technical test of the given artifact and specified goals along the way.

Design as a search process requires that the search for an effective artifact needs to utilize the available means to reach an end. This research started by looking at the current implementation and identified a need for tamper evident logging of the rebuild submissions.

Communication of the research means that the research needs to be communicated and presented to management and technical people. This research builds on top of the needs of the Debian community and solves problems directly related to the ever increasing threat of software supply chain security. We have also published the source code for this project to GitHub.

7.2 Technical Implementation

After each iteration of this project we have done testing to make sure the goals are met and to figure out if there has been any implementation bugs. This has resulted in utility tools to verify the integrity of the data structure, which was used in the technical testing.

Peffers et al. in "Design Science Research Evaluation" did a literature review over design science artifacts, and the evaluation method used [52], and described in Section 4.3 on page 28. Technical evaluations are done to assess how good the artifact is at solving the problem at hand, and is a suitable evaluation method when the artifact are software implementations.

The technical evaluation was done in Chapter 6 on page 71. The limitations in the implementation restricted the desire to replay larger data from the Debian build submissions. With more time the assessment and the technical evaluation could have been broader and covered a wider range of data.

There are some further improvements in the current data storage scheme we could consider. One of the main disadvantages in the current implementation is that all data is stored on one table for the Merkle tree. As PostgreSQL is poorly optimized for such workloads other ways to partition up the data model needs to be considered for future work. Other ways to layout the storage is to have one storage for interior nodes and the hashes of the leafs, and storage the leaf data in another table. This could potentially introduce less overheard when querying the Node table multiple times.

Another advanced strategy that could help the overall performance is to store complete subtrees in another table. This would shorten the time we have to look through the Node table to create the appropriate path. It would be a bit more difficult to implement, as we would need to keep

track of the tree height, and promote the largest complete tree for any given to a database for easy look up.

It should also be noted that there is a feature in the current implementation that is missing. Log monitors are used to validate that the elements on the transparency log are included, and the log is not being actively tampered with and issues consistency proofs. They are an essential part of a live implementation of a logging system. However, because of lack of time we did not manage to implement this feature. We are however testing the implementation in our own conditions. A log monitor is not needed to do so, and does not affect the proofs when testing the artifact.

During the development of this artifact, there was a realization that Google had implemented Trillian, which is a "... transparent, highly scalable and cryptographically verifiable data store" [27]. This is a free and open-source software library to implement verifiable data storage for things like certificate transparency logs, discussed in Section 2.5.1 on page 13.

Trillian is used in multiple certificate transparency log implementations, such as Nimbus from Cloudflare [11], and could prove a viable backend instead of a naive implementation. Interestingly, the implementation of Trillian is also supported by multiple SQL storage for things such as tree heads, leafs and subtrees. It would be interesting to investigate how such improvements would affect the performance of the current implementation.

7.3 Security Implications

The current implementation is by no means effective at what it does. But we can still assess the security implications of the project. The project properly implements the audit proof, and consistency proof as described in Section 5.2.2 on page 49

In Section 5.2.5 on page 68 we implement the integration into the APT package manager for Debian and verify the proofs locally without relying on the remove transparency server.

What this means is that we now have implemented new security features in the rebuilder. The usage of transparency logs, as described by Crosby and Wallach in "Efficient Data Structures for Tamper-Evident Logging" transparency logs enables us to detect tampering [15]. Before this rewrite, a visualizer can modify rebuild attestations with no sign of actually doing so. They can provide falsified information and trick APT into thinking it has gotten a valid package, but instead gotten a malicious one.

With our rewrite, we can now detect if tampering has happened by issuing the correct consistency and audit proofs to the log. This prevents the rebuilders from actively modifying content on the log.

7.4 Reproducible Research

Since we are after all writing about reproducible builds, trying to achieve reproducible research in this thesis seemed like a fitting goal. The evaluations and testing done in this project can be found in Appendix C on page 99.

Appendix A on page 95 lists the source code used in this project, and Appendix Section A.1 lists the source code for the transparency log implementation. The repository contains the data from the evaluation with the needed scripts to recreate any graphs in this thesis.

7.5 Research Questions

In this section we are going to take a look at the research questions defined in Section 1.3 on page 3 and answer them from the discussion in this chapter.

Research Question 1:

Can a transparency log provide additional security guarantees, and if so, how?
As discussed in Section 7.3 we can see that a transparency log indeed provides additional security guarantees on top of the previous rebuild setup. We are able to provide proofs that can prove whether or not tampering has taken place. This gives users the additional security guarantee of being able to detect tampering of the published build submissions.

Research Question 2:

Are we able to implement this into the current rebuilder verification process?
Section 5.2.5 on page 63 describes the integration into the APT transport that does the rebuilder verification process. We managed to implement this cleanly into the existing code base with no modification in the process itself.

Research Question 3:

Can this be deployed in a real-world scenario?
The following discussion in Section 7.2 shows that there are a few problems with the implementation in this thesis which makes it unfit for a real-world scenario. Chapter 6 on page 71 shows

how the performance of the artifact is not sufficient to meet the data demands from Debian packaging.

Chapter 8

Conclusion

This chapter presents and summarized the thesis. We will go through the research contributions, the limitations of the work we have done and the possible direction this project can take in the future.

8.1 Summary

In this thesis we have used Design Science research to develop a replacement for the API front end used by the rebuilder infrastructure, called visualizer. We have replaced the old implementation with a new extended version with support for additional security properties by implementing a transparency log on top of the API. We are capable of adding build submissions from the rebuilder, which consists of a BUILDINFO file and a in-toto link metadata file. We are capable of verifying whether or not published submissions have been tampered with after the fact and integrate with the Debian package manager.

The development has gone over four iterations by establishing a set of goals and making sure the goals are met after each iteration. After the iterations on the visualizer rewrite, we had an integration phase into the Debian package manager APT, to make sure we are able to utilize with the new API.

Before the first iteration of the project we established the requirements of the old API front end of the system. There where several API endpoints that had to be implemented to ensure backwards compatibility. This was the first iteration which was centered around making sure the code base is extensible and that the old API endpoints was correctly implemented.

The second iteration implemented the Merkle tree data structure in a PostgreSQL data storage back end. The implementation encompasses a dedicated API to investigate and debug the API,

along with appending raw JSON data structures to the tree it self. The proofs we need to prove whether or not the submission have been tampered with was implemented in this iteration: audit proofs and consistency proofs. The visualizer also got an endpoint to visualize the Merkle tree as a graph using the Graphviz library. The end result of this iteration was a working Merkle tree implementation that works as a transparency log and provide tamper evident logging. We are also capable of appending new elements without having to re-hash the interior nodes of the tree.

The third iteration centered around the cryptographic parts of the Merkle tree. All Merkle tree roots needs to be signed by a key pair to prove the given tree root comes from the given log server. This implemented the integration of a simple yet powerful library to aid in creating keys, signatures and the needed API endpoints of the crypto API. We also made sure all new tree roots are signed when appended leafs are added to the tree.

The fourth iteration implemented the transparency log overlay. We are for all intents and purposes implementing logging of events, and these carry some meaning which needs to be conveyed to the client. We implemented two kinds of overlay entries for package submission inclusion, and revocation. This allows the client to distinguish the meaning of different logged events. We are capable of having multiple package submissions on the log, in the form of inclusions, and revoke them if they where published or built erroneously. By appending BUILDINFO files, which describes the build environment of packages, and in-toto metadata, which is used to verify the integrity of the supply chain process, we are able to retrieve and verify packages on the client side. The result of this is an abstracted API for the client to utilize when verifying packages on the client side.

The final development section deals with the APT integration. APT is the Debian package manager and is used to download packages from remote repositories. By utilizing the transport system, and extending the transport written for the rebuilder system, we where able to extend and improve on the transport to utilize the new API and fetch in-toto link metadata from this process. We implemented the needed extensions to the transport so we could validate the consistency of rebuilders, and utilize the audit proofs of the fetched entries.

The evaluation of the prototype was done through technical testing where we appended entries to the log and tested the response time as the process went on. What we discovered is that the performance is not good enough for real-world use as the current data storage does not scale well enough. However we are capable of validate the proofs in the data storage and thus able of provide additional security guarantees to the APT integration.

The research has shown the transparency log does provide additional security guarantees, and that we are capable of implementing this into the current verification process. However the current implementation is not sufficient for real-world usage.

8.2 Future Work

The future work concerns the technical implementation of this project. Moving the back end implementation away from the homegrown SQL data storage this thesis presents, into a general data storage such as Trillian from Google would give valuable insight into the possible scalability of this technology. Other investigations into improvements of the data storage would be valuable to enhance and contribute to the software supply chain security of Debian packages.

8.3 Conclusion

This research has presented the theory, development and evaluation of a transparency log for a package rebuilder system. The research concludes that even if the current implementation is not sufficient for real-world usage, we are capable of having additional security guarantees with this project. However, improving the efficiency of this system needs to be explored further.

Bibliography

[1] Mustafa Al-Bassam and Sarah Meiklejohn. “Contour: A Practical System for Binary Transparency”. In: (Dec. 2017). eprint: 1712.08427v2. URL: http://arxiv.org/abs/1712.08427v2.

[2] Reproducible Builds. “Definitions”. In: (2019). URL: https://reproducible-builds.org/docs/definition/.

[3] Reproducible Builds. “Paris 2018”. In: (2018). URL: https://reproducible-builds.org/events/paris2018/.

[4] Reproducible Builds. “Recording the builds environment”. In: (2019). URL: https://reproducible-builds.org/docs/recording/.

[5] Reproducible Builds. “Sharing certifications”. In: (2019). URL: https://reproducible-builds.org/docs/sharing-certifications/.

[6] Reproducible Builds. “SOURCE_DATE_EPOCH”. In: (2019). URL: https://reproducible-builds.org/docs/source-date-epoch/.

[7] Reproducible Builds. “Who is involved?” In: (2019). URL: https://reproducible-builds.org/who/.

[8] R. Torres Cabrera and Bonnie Lee Appleton. “Software Reconstruction: Patterns for Reproducing Software Builds”. In: *Proceedings of the 6th Pattern Languages of Programs*. PLoP’99. 1999.

[9] Justin Cappos et al. “A Look in the Mirror: Attacks on Package Managers”. In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS ’08. Alexandria, Virginia, USA: ACM, 2008, pp. 565–574. ISBN: 978-1-59593-810-7. DOI: 10.1145/1455770.1455841. URL: http://doi.acm.org/10.1145/1455770.1455841.

[10] Stephen Checkoway et al. “A Systematic Analysis of the Juniper Dual EC Incident”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: ACM, 2016, pp. 468–479. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978395. URL: http://doi.acm.org/10.1145/2976749.2978395.

[11] Cloudflare. *Introducing Certificate Transparency and Nimbus*. 2018. URL: https://blog.cloudflare.com/introducing-certificate-transparency-and-nimbus/.

[12] Symantec Corporation. "Internet threat security report, 2018". In: *Symantec Corporation* (Apr. 1, 2018). URL: https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf (visited on 05/15/2019).

[13] Symantec Corporation. "Internet threat security report, 2019". In: *Symantec Corporation* (Feb. 1, 2019). URL: https://www.symantec.com/content/dam/symantec/docs/reports/istr-24-2019-en.pdf (visited on 05/15/2019).

[14] Russ Cox and Filippo Valsorda. "Proposal Secure the Public Go Module Ecosystem with the Go Notary". In: (2019). URL: https://go.googlesource.com/proposal/+/master/design/25530-notary.md.

[15] Scott A. Crosby and Dan S. Wallach. "Efficient Data Structures for Tamper-Evident Logging". In: *Presented as part of the 18th USENIX Security Symposium (USENIX Security 09)*. Montreal, Canada: USENIX, 2009. URL: https://www.usenix.org/node/.

[16] CrowdStrike. "Securing the supply chain". In: *CrowdStrike* (July 1, 2018). URL: https://www.crowdstrike.com/resources/wp-content/brochures/pr/CrowdStrike-Security-Supply-Chain.pdf (visited on 05/15/2019).

[17] Daniel Stenberg. *Command line tool and library for transferring data with URLs*. 1996. URL: https://curl.haxx.se/.

[18] Debian. *A Brief History of Debian. Chapter 4 - A Detailed History*. 2017. URL: https://www.debian.org/doc/manuals/project-history/ch-detailed.en.html.

[19] Debian. *deb-buildinfo - Debian build information file format*. 2018. URL: https://manpages.debian.org/stretch/dpkg-dev/deb-buildinfo.5.en.html.

[20] Roger Dingledine, Nick Mathewson, and Paul Syverson. "Tor: The Second-generation Onion Router". In: *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*. SSYM'04. San Diego, CA: USENIX Association, 2004, pp. 21–21. URL: http://dl.acm.org/citation.cfm?id=1251375.1251396.

[21] Aline Dresch, Daniel Pacheco Lacerda, and Jos Antnio Valle Antunes. *Design Science Research: A Method for Science and Technology Advancement*. Springer Publishing Company, Incorporated, 2014. ISBN: 3319073737, 9783319073736.

[22] RJ Ellison. "Evaluating and Mitigating Software Supply Chain Security Risks". In: (2010).

[23] John Ellson et al. "Graphviz — open source graph drawing tools". In: *Lecture Notes in Computer Science*. Springer-Verlag, 2001, pp. 483–484.

[24] Foteinos Mergoupis. *pymerkle*. 2018. URL: https://github.com/FoteinosMerg/pymerkle.

[25] The Linux Foundation. *Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It*. 2015. URL: https://www.linux.com/publications/linux-kernel-development-how-fast-it-going-who-doing-it-what-they-are-doing-and-who.

[26] Gentoo Linux. *Project:Infrastructure/Incident Reports/2018-06-28 Github*. 2018. URL: https://wiki.gentoo.org/wiki/Project:Infrastructure/Incident_Reports/2018-06-28_Github.

[27] Google. *A transparent, highly scalable and cryptographically verifiable data store.A transparent, highly scalable and cryptographically verifiable data store*. 2016. URL: https://github.com/google/trillian.

[28] Alan R. Hevner et al. "Design Science in Information Systems Research". In: *MIS Q.* 28.1 (Mar. 2004), pp. 75–105. ISSN: 0276-7783. URL: http://dl.acm.org/citation.cfm?id=2017212.2017217.

[29] Benjamin Hof and Georg Carle. "Software Distribution Transparency and Auditability". In: (Dec. 2017). eprint: 1711.07278v1. URL: http://arxiv.org/abs/1711.07278v1.

[30] Paul E. Hoffman and Bruce Schneier. *Attacks on Cryptographic Hashes in Internet Protocols*. RFC 4270. Dec. 2005. DOI: 10.17487/RFC4270. URL: https://rfc-editor.org/rfc/rfc4270.txt.

[31] Holger Levsen and many others. *Overview of various statistics about reproducible builds*. 2014. URL: https://tests.reproducible-builds.org/debian/reproducible.html.

[32] J. D. Hunter. "Matplotlib: A 2D Graphics Environment". In: *Computing in Science Engineering* 9.3 (May 2007), pp. 90–95. ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.55.

[33] UH-IaaS. *A NORWEGIAN CLOUD INFRASTRUCTURE FOR RESEARCH AND EDUCATION*. 2018. URL: http://www.uh-iaas.no/.

[34] Bala Iyer and George Wyner. "Evaluating APIs: A Call for Design Science Research". In: *Proceedings of the 7th International Conference on Design Science Research in Information Systems: Advances in Theory and Practice*. DESRIST'12. Las Vegas, NV: Springer-Verlag, 2012, pp. 28–35. ISBN: 978-3-642-29862-2. DOI: 10.1007/978-3-642-29863-9_3. URL: http://dx.doi.org/10.1007/978-3-642-29863-9_3.

[35] Jared K. Smith. *Security incident on Fedora infrastructure on 23 Jan 2011*. 2011. URL: https://lists.fedoraproject.org/pipermail/announce/2011-January/002911.html.

[36] Juniper. “2015-12 Out of Cycle Security Bulletin: ScreenOS: Multiple Security issues with ScreenOS (CVE-2015-7755, CVE-2015-7756)”. In: *Juniper* (Dec. 17, 2015). URL: https://kb.juniper.net/InfoCenter/index?page=content&id=JSA10713 (visited on 05/15/2019).

[37] Chris Lamb. “buildinfo.debian.net - A proof-of-concept .buildinfo server”. In: (2015). URL: https://buildinfo.debian.net/.

[38] Ben Laurie, Adam Langley, and Emilia Kaster. “RFC 6962 - Certificate Transparency”. In: (2013). URL: https://tools.ietf.org/html/rfc6962.

[39] Linus Torvalds. *Git - fast, scalable, distributed revision control system*. 2005. URL: https://github.com/git/git.

[40] S. Lipner. “The Trustworthy Computing Security Development Lifecycle”. In: *20th Annual Computer Security Applications Conference*. None 2004, pp. 2–13. DOI: 10.1109/CSAC.2004.41.

[41] Lunar. *Byte-for-byte identical reproducible builds? - BoF at DebConf13*. 2013. URL: https://wiki.debian.org/ReproducibleBuilds/History?action=AttachFile&do=view&target=dc13-bof-reproducible-builds.txt.

[42] H Steiner M Perry S Schoen. “Reproducible Builds. Moving Beyond Single Points of Failure for Software Distribution”. In: (2014).

[43] Wes McKinney. *Data Structures for Statistical Computing in Python*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 51–56.

[44] Dirk Merkel. “Docker: Lightweight Linux Containers for Consistent Development and Deployment”. In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=2600239.2600241.

[45] Ralph C. Merkle. “A digital signature based on a conventinal encryption function”. In: (1998).

[46] Miron Cuperman. *Gitian: a secure software distribution*. 2009. URL: https://gitian.org/.

[47] New York University - Secure Systems Lab. *Cryptographic and general-purpose routines for Secure Systems Lab projects at NYU*. 2016. URL: https://github.com/secure-systems-lab/securesystemslib.

[48] Kirill Niktin. “CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchain and Verified Builds”. In: (2017).

[49] Hayden Parker. *Analysis of a Supply Chain Attack*. 2018. URL: https://medium.com/@hkparker/analysis-of-a-supply-chain-attack-2bd8fa8286ac.

[50] Paul W. Frields. *Infrastructure report, 2008-08-22 UTC 1200*. 2008. URL: https://www.redhat.com/archives/fedora-announce-list/2008-August/msg00012.html.

[51] Christina Paule, Thomas Düllmann, and André van Hoorn. "Vulnerabilities in Continuous Delivery Pipelines? A Case Study". In: *Proceedings of the 2019 IEEE International Conference on Software Architecture (ICSA 2019)*. Mar. 2019. DOI: 10.1109/ICSA-C.2019.00026.

[52] Ken Peffers et al. "Design Science Research Evaluation". In: *Proceedings of the 7th International Conference on Design Science Research in Information Systems: Advances in Theory and Practice*. DESRIST'12. Las Vegas, NV: Springer-Verlag, 2012, pp. 398–410. ISBN: 978-3-642-29862-2. DOI: 10.1007/978-3-642-29863-9_29. URL: http://dx.doi.org/10.1007/978-3-642-29863-9_29.

[53] Mike Perry. "[liberationtech] Deterministic builds and software trust [was: Help test Tor Browser!]" In: (2013). URL: https://mailman.stanford.edu/pipermail/liberationtech/2013-June/009257.html.

[54] PostgreSQL Global Development Group. *PostgreSQL*. 2008. URL: http://www.postgresql.org.

[55] Python Core Team. *Python: A dynamic, open source programming language. Python Software Foundation*. 2015. URL: https://www.python.org/.

[56] Armin Ronacher. *Flask*. 2010. URL: http://flask.pocoo.org/.

[57] Armin Ronacher. *jinja2*. 2007. URL: http://jinja.pocoo.org/.

[58] Santiago Torres-Arias. *in-toto: A framework to secure the integrity of the software supply chain*. 2016. URL: https://in-toto.github.io/.

[59] SQLAlchemy authors and contributors. *SQLAlchemy: The Python SQL Toolkit and Object Relational Mapper*. 2005. URL: https://www.sqlalchemy.org/.

[60] Liran Tal. *Malicious remote code execution backdoor discovered in the popular bootstrap-sass Ruby gem*. 2019. URL: https://snyk.io/blog/malicious-remote-code-execution-backdoor-discovered-in-the-popular-bootstrap-sass-ruby-gem/.

[61] The Linux Foundation. *The Cracking of Kernel.org*. Aug. 2011. URL: https://www.linuxfoundation.org/blog/2011/08/the-cracking-of-kernel-org/.

[62] The Tor Project. *The Tor Browser*. 2002. URL: https://www.torproject.org/download/.

[63] Ken Thompson. "Reflections on Trusting Trust". In: *Commun. ACM* 27.8 (Aug. 1984), pp. 761–763. ISSN: 0001-0782. DOI: 10.1145/358198.358210. URL: http://doi.acm.org/10.1145/358198.358210.

[64] Vijay K. Vaishnavi and William Kuechler. *Design Science Research Methods and Patterns: Innovating Information and Communication Technology, 2Nd Edition*. 2nd. Boca Raton, FL, USA: CRC Press, Inc., 2015. ISBN: 1498715257, 9781498715256.

[65] Edgar Weippl et al. "Transparency Overlays and Applications". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*. None 2016, pp. 168–179. DOI: 10.1145/2976749.2978404.

[66] David Wheeler. "Countering Trusting Trust Through Diverse Double-Compiling". In: *Proceedings of the 21st Annual Computer Security Applications Conference*. ACSAC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 33–48. ISBN: 0-7695-2461-3. DOI: 10.1109/CSAC.2005.17. URL: https://doi.org/10.1109/CSAC.2005.17.

Appendix A

Development source links

A.1 Source Code for master project

https://github.com/Foxboron/master

A.2 APT Transport Source Code

The original source of the APT transport
https://github.com/in-toto/apt-transport-in-toto

The fork of the source containing the transparency overlay implementation
https://github.com/Foxboron/apt-transport-in-toto/commit/6e529959dc20932efd573ad5b4f803d

A.3 buildinfo.debian.net pull-request

https://github.com/lamby/buildinfo.debian.net/pull/54

A.4 Rebuilder Source Code

Main repository for the rebuilder source code
https://salsa.debian.org/reproducible-builds/debian-rebuilder-setup/

Fork of the main repository for the UH-IaaS rebuilder
https://salsa.debian.org/Foxboron-guest/debian-rebuilder-setup

Appendix B

APT Testing setup

To run the APT transport test suite you need to have a running docker daemon and the correct privileges set up. Below is the expected abbreviated output of the test run.

```
$ git clone https://github.com/Foxboron/master.git
Cloning into 'master'...
$ cd master
$ make transport-test
[*] Starting
[*] Waiting for system to start...
[*] Adding test data...
[*] Running test suite...
[...]
apt_1       | py37 create: /app/.tox/py37
apt_1       | py37 installdeps: -rrequirements.txt, pylint, bandit, coverage, mock
apt_1       | py37 installed: asn1crypto==0.24.0,astroid==2.2.5,attrs==19.1.0,bandit==1.
apt_1       | py37 run-test: commands[2] | coverage run -m unittest discover
[...]
apt_1       | ----------------------------------------------------------------------
apt_1       | Ran 11 tests in 18.749s
apt_1       | OK
[...]
apt_1       |   py37: commands succeeded
apt_1       |   congratulations :)
[...]
[*] Removing containers...
```

Appendix C

Evaluation

C.1 Merkle tree stress test

```
$ git clone https://github.com/Foxboron/master.git
Cloning into 'master'...
$ cd master
$ make stress-test
[*] Starting
...
[*] Waiting for system to start...
[*] Adding 113054 elements to the tree - THIS WILL TAKE A WHILE...
make: Entering directory '/app'
(cd test; python create_tree.py --new --append 113054)
make: Leaving directory '/app'
[*] Copying stats.txt and roots.txt locally...
[*] Removing containers...
...
```

Listing 29: Running the stress test

Listing 29 shows the setup for the stress test in the evaluation. Since replicating the testing in complete, with 113054 elements takes days to complete, a smaller version with “small-stress-test” has been added for the sake of saving time.

```
$ python stats/plot_package_count.py stats.txt
$ python stats/plot_response_time.py stats.txt
```

Listing 30: Plotting the graphs

Listing 30 shows how to generate the graphs in the evaluation. The “stats.txt” file contains the number of leafs, interior nodes, hash and response time on the query, and will be generated when the stress tests are done from Listing 29. The data used in the evaluation can be found in “stats/stats.txt”.

C.2 BUILDINFO FTP Server

https://buildinfos.debian.net/ftp-master.debian.org/buildinfo/2019/