ABSTRACT

TOWARDS ENSURING INTEGRITY AND AUTHENTICITY OF SOFTWARE REPOSITORIES

by
Sangat Vaidya

The software development process comprises a series of steps known as a software supply chain. These steps include managing the source code, testing, building and packaging it into a final product, and distributing the product to end users. Along this chain, software repositories are used for different purposes such as source code management (Git, SVN, mercurial), software distribution (PyPI, RubyGems, NPM) or for deploying software based on container images (Harbor, DockerHub, Artifact Hub). In the recent past, different types of repositories have increasingly been the target of attacks. As such, there is a need for mechanisms to ensure integrity and authenticity of repository data. This work seeks to design mechanisms for providing end users with integrity and authenticity guarantees for repositories used in the software development process.

In the first part of this work, the focus is on version control systems that are used by software developers for software code management and collaboration. Recent history has shown that source code repositories represent appealing attack targets. Attacks that violate the integrity of repository data can impact negatively millions of users. This work designs and implements a commit signing mechanism for centralized version control systems that rely on a client-server architecture. When the proposed commit signing protocol is in place, the integrity and authenticity of the repository can be guaranteed even when the server hosting the repository is not trustworthy.

The second part of this work proposes an approach for certifying the validity of software projects hosted on community repositories. This work designs and implements a Software Certification Service (SCS) that receives certification requests

from a project owner for a specific project and then issues a project certificate once the project owner successfully completes a protocol for proving ownership of the project. The proposed certification protocol is inspired from the highly-successful ACME protocol used by the Let's Encrypt certification authority and can be fully automated on the SCS side. However, it is fundamentally different in its attack mitigation capabilities and in how ownership is proven. It is also compatible with existing community repositories such as PyPI, RubyGems, NPM, or GitHub, without requiring any changes to these repositories. To support the claim, the work instantiates the proposed certification service with several practical deployments.

In the last part of this work, the focus is on artifact repositories that are used for deployment of software. These repositories are used to manage deployment artifacts such as container images, Helm charts and policy bundles. Current artifact management systems lack proper version control features. This work proposes a uniform version control system for such artifacts. The primary focus here is on artifacts recognized by the Open Container Initiative (OCI) standards. The approach treats artifacts as structured objects with multiple components such as file systems, binary packages, and metadata, instead of treating them as just opaque binary objects. The work further leverages this structure to design a diff algorithm that computes the difference between two versions of artifacts. The approach examines challenges related to computing differences between versions, the persistence of older versions of artifacts, and the security aspects of a version controlling system. Finally, the work proposes commit and update mechanisms for version control that address these challenges. With the proposed commit and update protocols in place, various types of OCI artifacts can be version controlled uniformly, regardless of their types.

TOWARDS ENSURING INTEGRITY AND AUTHENTICITY OF SOFTWARE REPOSITORIES

by
Sangat Vaidya

A Dissertation Proposal
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

August 2022

APPROVAL PAGE

TOWARDS ENSURING INTEGRITY AND AUTHENTICITY OF SOFTWARE REPOSITORIES

Sangat Vaidya

Reza Curtmola, Dissertation Advisor — Date
Professor of Computer Science, NJIT

Cristian Borcea, Committee Member — Date
Professor of Computer Science, NJIT

Vincent Oria, Committee Member — Date
Professor of Computer Science, NJIT

Kurt Rohloff, Committee Member — Date
Associate Professor of Computer Science, NJIT

Justin Cappos, Committee Member — Date
Associate Professor of Computer Science and Engineering, New York University

BIOGRAPHICAL SKETCH

Author: Sangat Vaidya

Degree: Doctor of Philosophy

Date: August 2022

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science
  New Jersey Institute of Technology, NJ, USA, 2022
- Master of Technology in Computer Science,
  National Institute of Technology, Calicut, India, 2016
- Bachelor of Engineering in Computer Engineering,
  Gujarat Technological University, Ahmedabad, India, 2014

Major: Computer Science

Presentations and Publications:

Vaidya, S., Torres-Arias, S., Curtmola, R., and Cappos, J. Bootstrapping trust in community repository projects. To appear In Proceedings of the 18th EAI International Conference on Security and Privacy in Communication Networks (SecureComm '22), Springer International Publishing.

Vaidya, S., Torres-Arias, S., Curtmola, R., and Cappos, J. Commit signatures for centralized version control systems. In ICT Systems Security and Privacy Protection (2019), Springer International Publishing, pp. 359-373.

This work is dedicated to H.D.H. Hariprasad Swamiji Maharaj

ACKNOWLEDGMENT

I extend my sincerest gratitude to my dissertation advisor, Professor Reza Curtmola, for his invaluable guidance, moral support, patience and encouragement and for the opportunity to conduct research in the fast developing field of software supply chain security.

Special thanks are due towards Professor Justin Cappos who not only served as a committee member but has also been a very supportive mentor throughout this journey. Furthermore, I extend my gratitude to thank Professors Cristian Borcea, Vincent Oria and Kurt Rohloff for serving as committee members and taking time to review the work.

I also extend my gratitude towards the National Science Foundation (NSF) and the Defense Advanced Research Projects Agency (DARPA), without whose support the research would not have been conducted. This work was supported by the NSF grant CNS 1801430 and the DARPA Contract No. A8650-15-C-7521. I would also like to express special thanks to my co-author Professor Santiago Torres-Arias for his valuable input during every stage of this research work.

I would like to thank P.P. Premswarup Swamiji, P.P. Tyagvallabh Swamiji and everyone in Haridham mandir for their constant support in various ways throughout the journey.

Last but not least, my sincere thanks to my parents, Dr. Samir Vaidya and Soochi Vaidya and everyone in the family for their support, understanding and encouragement throughout my research work. I would like to thank my sister, Sahaj Vaidya for making the journey joyful throughout. I would like to extend my gratitude to Cavidan Yakupoglu and Archita Zaveri for always being there in times of need. A special thanks to Dr. Jyot Buch for support during the last mile of this journey.

TABLE OF CONTENTS

Chapter **Page**

1 INTRODUCTION 1

2 ENSURING INTEGRITY OF SOURCE CODE REPOSITORIES 5

2.1 Introduction 5

2.2 Background 8

2.2.1 Centralized Version Control Systems 9

2.2.2 Merkle Hash Trees 12

2.3 Can Git Commit Signing be Used? 14

2.4 Adversarial Model and Security Guarantees 17

2.4.1 Attacks 19

2.4.2 Security Guarantees 20

2.5 Commit Signatures for Centralized VCS-es 21

2.5.1 Secure Commit Protocol 21

2.5.2 Secure Update Protocol 22

2.5.3 MHT-based Proofs 23

2.6 Security Analysis (Sketch) 26

2.7 Implementation and Experimental Evaluation 27

2.7.1 Implementation 27

2.7.2 Experimental Setup 29

2.7.3 Experimental Evaluation for Commit Operations 30

2.7.4 Experimental Evaluation for Update Operations 32

2.8 Related Work 34

2.9 Conclusion 35

3 BOOTSTRAPPING TRUST FOR SOFTWARE REPOSITORIES 37

3.1 Introduction 37

3.2 Background 41

TABLE OF CONTENTS
(Continued)

Chapter **Page**

3.3 Existing Mechanisms 43
3.3.1 Code Signing 43
3.3.2 Package Signatures 45
3.3.3 Binary Transparency 46
3.4 System and Threat Model 47
3.4.1 System Model 47
3.4.2 Threat Model and Security Goals 49
3.5 Protocol 51
3.5.1 Preliminaries 51
3.5.2 Certification Protocol Description 54
3.6 Security Analysis 59
3.6.1 Compromise Resiliency 61
3.7 Deployment 63
3.7.1 SCS Implementation Details 63
3.7.2 Deployment to Community Repositories 64
3.7.3 Automating Delegations in Community Repositories 67
3.8 Related Work 69
3.8.1 Securing Community Repositories 69
3.8.2 Software Certification 70
3.9 Conclusion 70
4 VERSION CONTROL SYSTEM FOR ARTIFACTS 72
4.1 Introduction 72
4.2 Background 75
4.2.1 Open Container Initiative (OCI) and OCI Artifacts 75
4.2.2 How are Artifacts Related? 77
4.3 Benefits of VCS for Artifacts 78

TABLE OF CONTENTS
(Continued)

Chapter **Page**

4.4 Challenges in Designing a VCS for Artifacts 79
4.5 Current Practices for Versioning in OCI Artifacts 81
4.5.1 Versioning Using Existing Version Control Systems 82
4.5.2 Versioning Using Configuration Files 83
4.5.3 Versioning Using Tags 85
4.6 Internal Structure of the Harbor Registry 86
4.6.1 Storage Structure of Harbor 86
4.6.2 Data Structures in Harbor 87
4.6.3 Creation and Deletion of Blobs 88
4.6.4 Security Features in Harbor 89
4.7 Internal Structure of Proposed VCS 89
4.7.1 Storage Structure 89
4.7.2 Security Features in VCS 91
4.7.3 Commit and Update Protocols 92
4.7.4 Artifacts as Structured Objects 93
4.7.5 Diff Algorithm 95
4.8 Discussion 97
4.9 Conclusion 98
5 CONCLUSION 100
BIBLIOGRAPHY 103

LIST OF TABLES

Table **Page**

2.1 Statistics for the Selected Repositories (as of March 2018). The number of Files and the Average File Size are Based on the Latest Revision in the Repository 29

2.2 Commit Time for One Revision (in seconds) 30

2.3 Server Storage for One Revision (in MBs) 31

2.4 Network Communication from Client to Server for Committing One Revision (in KBs) 31

2.5 Network Communication from Server to Client for Committing One Revision (in KBs) 31

2.6 Update Time for One Revision (in seconds) 33

2.7 Comparison Between SSVN and SSVN-skip for a Large Repository (the last 100 revisions in GCC) 33

2.8 Network Communication from Client to Server for Updating One Revision (in KBs) 33

2.9 Network Communication from Server to Client for Updating One Revision (in KBs) 34

LIST OF FIGURES

Figure **Page**

2.1 SVN follows a centralized repository model. The central SVN repository stores all revisions, whereas clients store only one revision. 10

2.2 A standard Merkle Hash Tree. The set of elements consists of A, B, C, D, E, F, G, H. We use the following notation: $H_A = h(A)$, and $H_{AB} = h(H_A||H_B)$, and $H_{ABCDEFGH} = h(H_{ABCD}||H_{EFGH})$, where h is a collision-resistant hash function and "||" denotes concatenation of strings. 13

2.3 A non-standard Merkle Hash Tree. 14

2.4 Git object hierarchy for a repository. 15

2.5 MHT for a revision of repository R1. 25

3.1 The CA asks the client to prove control over domain name *example.com*. 42

3.2 The CA verifies that the client has proven control over domain name *example.com*. 42

3.3 The software certification architecture. 47

3.4 Challenge response displayed in the "Project description" section for a PyPI package. 66

3.5 Challenge Response displayed on the project webpage for a RubyGems project. 66

3.6 Challenge response displayed in the Readme file for an NPM package. . 67

3.7 Challenge response displayed in the README.md file for a GitHub project. 68

4.1 Structure of Artifact Blob in Harbor. 88

4.2 Structure and computation of Project Blob in Harbor. 89

4.3 Structure of Commit Blob and computation of signature. 91

CHAPTER 1

INTRODUCTION

The software development process comprises a series of steps known as a software supply chain. Some of those steps include managing the source code, testing, building, and packaging software into a final product, and distributing the product to end users. During the development process, the developers use software repositories for different purposes like source code management (Git, SVN, mercurial), software distribution (PyPI, RubyGems, NPM) or repositories for container images used for deploying software (DockerHub, Amazon ECR, Google Container Repository). In the recent past, different types of repositories have increasingly been the target of attacks. Due to this reason, the repositories need mechanisms to ensure integrity and authenticity of the data.

In the course of this work, we seek to design mechanisms to provide end users with integrity and authenticity of repositories used in the software development process.

In the first part of this work, we focus on the version control systems (VCS-es) that are used by software developers for software code management and collaboration. VCS-es play a major role in the software development life cycle, yet historically their security has been relatively underdeveloped compared to their importance. Recent history has shown that source code repositories represent appealing attack targets. Attacks that violate the integrity of repository data can impact negatively millions of users. Some VCS-es, such as Git, employ commit signatures as a mechanism to provide developers with cryptographic protections for the code they contribute to a repository. However, an entire class of other VCS-es, including the well-known

Apache Subversion (SVN), lacks such protections and is susceptible to attacks that tamper with repository data in an undetectable fashion.

We design and implement a commit signing mechanism for centralized version control systems that rely on a client-server architecture [1]. Our solution is the first that supports VCS features such as working with a portion of the repository on the client side and allowing clients to work on disjoint sets of files without having to retrieve each other's changes. During a commit, clients compute the commit signature over the root of a Merkle Hash Tree (MHT) built on top of the repository. A client obtains from the server an efficient proof that covers the portions of the repository that are not stored locally, and uses it in conjunction with data stored locally to compute the commit signature. During an update, a client retrieves a revision's data from the central repository, together with the commit signature over that revision and a proof that attests to the integrity and authenticity of the retrieved data. To minimize the performance footprint of the commit signing mechanism, the proofs about non-local data contain siblings of nodes in the Steiner tree determined by the items in the commit/update changeset. In addition, the server stores the MHT-related metadata in an incremental fashion, using skip-delta encoding. This provides an advantageous tradeoff between storage and computation which significantly reduces the server-side storage overhead for large repositories. When our commit signing protocol is in place, the integrity and authenticity of the repository can be guaranteed even when the server hosting the repository is not trustworthy.

In the second part of this work, we focus on the problem of bootstrapping trust in a piece of software. Open source software packages are very popular among developers. These packages are distributed using community repositories. Since these community repositories are very popular, they are also targets of attacks. Currently, the community repositories do not have sufficient mechanisms to ensure authenticity and integrity of the software that users download from these repositories. In this

work, we propose to design and implement a Software Certification Service (SCS) that receives certification requests from a project owner for a specific project and then issues a project certificate once the project owner successfully completes a procedure for proving ownership of the project. That certificate is then shipped with the software project and the end user uses it to bootstrap the verification of the software supply chain integrity.

The main problem that needs to be solved in this context is proving the ownership of a software project, *i.e.*, how can the owner of a software project convince a certification authority that it owns a specific software project. We focus on proving ownership of open-source projects that are hosted on community repositories such as PyPI, RubyGems, NPM, or GitHub. Inspired from the protocol used by the Let's Encrypt certification authority [2] to issue free X.509 domain certificates for TLS encryption, we design and implement a service that allows the owner of a software project hosted on a community repository to obtain a software certificate for that project. The service does not require any changes to the community repositories where projects are hosted and, thus, can be deployed easily. The service relies on a protocol executed between a client (project owner) and a server (SCS), which allows the client to prove ownership of the project through its ability to write into the project's repository. To prevent a range of potential attacks, the protocol requires the owner to prove ownership over a longer period of time, in a publicly visible fashion, so that attempts by an unauthorized party to claim project ownership would be difficult to remain undetected. We use the service to automate the certification for software projects hosted on repositories like PyPI, RubyGems, NPM or GitHub. We also extend the service to automate delegation process in community repositories that use services like TUF [3, 4, 5] to provide compromise resilience.

In the last part of work, we focus on the security of artifact registries that are used for deployment of software projects. Software deployment process makes use

of several artifacts. Container images [6], Helm Charts [7], OPA policy bundles [8], Singularity images [9] are some of the artifacts that are used for various purposes during the software deployment process. During recent years, containers and related artifacts are gaining popularity for deployment of software. As a result, there has been a lot of research in various security aspects of these artifacts and registries used for artifact management. But, one major feature that these registries lack is proper version control.

The last chapter proposes a uniform version control system for the artifacts recognized by Open Container Initiative(OCI) standards. One major reason for absence of version control for these artifacts is that the registries treat them as binary files. As a result, when changes are made to these artifacts and pushed to the registries, the changes are tracked in terms of bytes and artifacts are treated as binary objects. This makes it difficult to track changes between two versions. Hence, the absence of version control systems. In the proposed approach we treat artifacts as structured objects with multiple components like file systems, binary packages, and metadata, instead of treating them as just opaque binary objects. We further leverage this structure to design a diff algorithm that computes the difference between two versions of artifacts. We also examine challenges related to computing differences between the two versions, the persistence of older versions of artifacts, and the security aspects of version control system. Finally, we propose commit and update mechanisms for version control that address these challenges. With the proposed commit and update protocols in place, various types of OCI artifacts can be version controlled uniformly, regardless of their types.

CHAPTER 2

ENSURING INTEGRITY OF SOURCE CODE REPOSITORIES

Version Control Systems (VCS-es) play a major role in the software development life cycle, yet historically their security has been relatively underdeveloped compared to their importance. Recent history has shown that source code repositories represent appealing attack targets. Attacks that violate the integrity of repository data can impact negatively millions of users. Some VCS-es, such as Git, employ *commit signatures* as a mechanism to provide developers with cryptographic protections for the code they contribute to a repository. However, an entire class of other VCS-es, including the well-known Apache Subversion (SVN), lacks such protections.

In this chapter, we design the first commit signing mechanism for centralized version control systems, which supports VCS features such as working with a subset of the repository and allowing clients to work on disjoint sets of files without having to retrieve each other's changes. In our solution, developers rely on proofs from the server to ensure that the server provides correct information about data that is not stored locally. To minimize the performance footprint of the commit signing mechanism, the commit signature is based on a Merkle Hash Tree computed over the repository, whereas the proofs about non-local data contain siblings of nodes in the Steiner tree determined by items in the commit/update changeset. We implement a prototype for the proposed commit signing mechanism on top of the SVN codebase and show experimentally that it only incurs a modest overhead. When our solution is in place, the VCS security model is substantially improved.

2.1 Introduction

A Version Control System (VCS) plays an important part in any software development project. The VCS facilitates the development and maintenance process by allowing

multiple contributors to collaborate in writing and modifying the source code. The VCS also maintains a history of the software development in a source code repository, thus providing the ability to rollback to earlier versions when needed. Some well-known VCS-es include Git [10], Subversion [11], Mercurial [12], and CVS [13].

Source code repositories represent appealing attack targets. Attackers that break into repositories can violate their integrity, both when the repository is hosted independently, such as internal to an enterprise, or when the repository is hosted at a specialized provider, such as GitHub [14], GitLab [15], or Sourceforge [16]. The attack surface is even larger when the hosting provider relies on the services of a third party for storing the repository, such as a cloud storage provider like Amazon or Google. Integrity violation attacks can introduce vulnerabilities by adding or removing some part of the codebase. In turn, such malicious activity can have a devastating impact, as it affects millions of users that retrieve data from the compromised repositories. In recent years, these types of attacks have been on the rise [17], and have affected most types of repositories, including Git [18, 19, 20, 21], Subversion [22, 23], Perforce [24], and CVS [25].

To ensure the integrity and authenticity of externally-hosted repositories, some VCS-es such as Git and Mercurial employ a mechanism called *commit signatures*, by which developers can use digital signatures to protect the code they contribute to a repository. When signing using their private keys, developers will ensure that any tampering with the repository can be detected. Perhaps surprisingly, several other VCS-es, such as Apache Subversion (known as SVN), lack this ability and are thus more vulnerable to attack. We note that although the market share of SVN has declined, its security is still important to address because it is the second most popular among open source projects [26] and it is the de facto standard enterprise choice for many companies. Thus, lacking the ability to sign commits can have a considerable

impact, as it opens the possibility of file manipulation on a remote repository in an undetectable fashion.

Our goal is to enable commit signature functionality, and thus improve security, for a class of VCS-es that have a centralized client-server architecture and share a similar set of features. We describe a solution for SVN [11], which is representative for this class of VCS-es, but our techniques are applicable to other VCS-es such as GNU Bazaar [27], Perforce Helix Core [28], Surround SCM [29], StarTeam [30], and Vault [31].

Contributions. In this chapter, we design and implement a commit signing mechanism for centralized version control systems that rely on a client-server architecture. Our solution is the first that supports VCS features such as working with a portion of the repository on the client side and allowing clients to work on disjoint sets of files without having to retrieve each other's changes. During a commit, clients compute the commit signature over the root of a Merkle Hash Tree (MHT) built on top of the repository. A client obtains from the server an efficient proof that covers the portions of the repository that are not stored locally, and uses it in conjunction with data stored locally to compute the commit signature. During an update, a client retrieves a revision's data from the central repository, together with the commit signature over that revision and a proof that attests to the integrity and authenticity of the retrieved data. To minimize the performance footprint of the commit signing mechanism, the proofs about non-local data contain siblings of nodes in the Steiner tree determined by items in the commit/update changeset. In addition, the server stores the MHT-related metadata in an incremental fashion, using skip-delta encoding. This provides an advantageous tradeoff between storage and computation, significantly reducing server-side storage overhead for large repositories.

When our commit signing protocol is in place, repository integrity and authenticity can be guaranteed even when the server hosting the repository is not trustworthy. We make the following contributions:

- We examine Apache SVN, a representative centralized version control system, and identify a range of attacks that stem from the lack of integrity mechanisms for the repository.
- We identify fundamental architectural and functional differences between centralized and decentralized VCS-es. Decentralized VCS-es like Git replicate the entire repository at the client side and eliminate the need to interact with the server when performing commits. Moreover, they do not support partial checkouts and require clients to retrieve other clients' changes before committing their own changes. These differences introduce security and performance challenges that prevent us from applying to centralized VCS-es a commit signing solution such as the one used in Git.
- We design the first commit signing mechanism for centralized VCS-es that rely on a client-server architecture and support features such as working with a subset of the repository and allowing clients to work on disjoint sets of files without having to retrieve each other's changes. Our solution substantially improves the security model of such version control systems. Although we show this design in the context of Apache SVN, our solution is applicable to other VCS-es that fit this model [27, 28, 29, 30, 31].
- We implement SSVN, a prototype for the proposed commit signature mechanism on top of the SVN codebase. We perform an extensive experimental evaluation based on three representative SVN repositories (FileZilla, SVN, GCC) and show that SSVN is efficient and incurs only a modest overhead compared to a regular (insecure) SVN system.

2.2 Background

This section provides background on version control systems (VCS-es) that have a (centralized) client-server architecture [27, 28, 29, 30, 31] and on (non-standard) Merkle Hash Trees, which will be used in subsequent sections. We overview the main protocols of such VCS-es, commit and update, which have been designed for a benign setting (*i.e.*, the VCS server is assumed to be fully trusted). Our description is focused on Apache SVN [11], an open source VCS that is representative for this

class of VCS-es. Other centralized VCS-es fit this description, with small variations. Readers familiar with these concepts are advised to skip this section.

2.2.1 Centralized Version Control Systems

A version control system (VCS) manages source code files and directories and the changes made to them, over time. This allows users to keep track of changes to the data and recover older versions of the data. Thus, a VCS provides many useful features to software developers, such as retrieving previous versions of the source code in order to locate and fix bugs, rolling back to earlier versions in case the working version becomes corrupted, or allowing team development in which multiple developers can work simultaneously on updates.

In a centralized VCS, the VCS server stores the main repository for a project and multiple clients collaborate on the project. The main (central) repository contains all the revisions since the project was created, whereas each client stores in its local repository only one revision, referred to as a *base revision*. Typically, clients seek to retrieve the latest revision from the server as their base revision, but may choose to retrieve any other (older) revision. The clients make changes to their local repositories and then publish these changes in the central repository on the server for others to see these changes. Figure 2.1 shows an SVN server that stores a project with 100 revisions, and four clients, each of which stores a specific revision.

Project management involves two components: the main repository on the server side and a *local working copy (LWC)* on the client side. The LWC contains a *base revision* for files retrieved by the client from the main repository, plus any changes the client makes on top of the base revision.

A client can publish the changes from her LWC to the main repository by using the "`commit`" command. As a result, the server creates a new revision which incorporates these changes into the main repository. If a client wants to update her

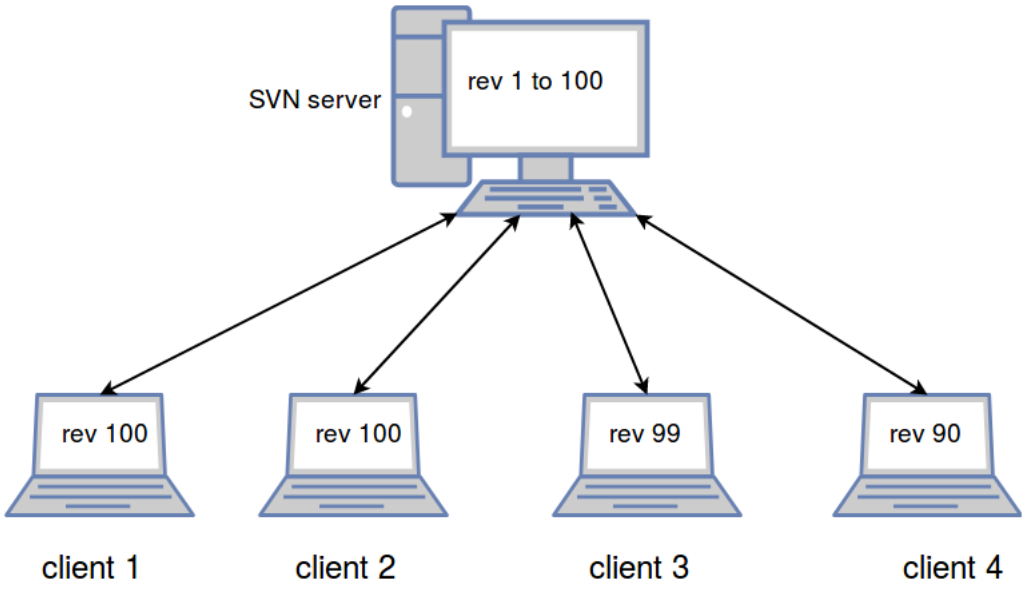


Figure 2.1 SVN follows a centralized repository model. The central SVN repository stores all revisions, whereas clients store only one revision.

LWC with the changes made by other clients, she uses the "`update`" command; this fetches the latest revision from the server.

The codebase revisions are referred to by a unique identifier called a *revision number*. In SVN, this is an integer number that has value 1 initially and is incremented by 1 every time a client commits changes to the repository. A revision in the SVN repository refers to the entire repository and not just to the files and directories changed in that revision. For example, whenever the server creates a new revision, all the files and directories in the repository are assigned that new revision number. It should be noted that files in the client's LWC does not have to correspond to any single revision, as the client may have a mix of files and directories belonging to different revision numbers. This is possible because SVN allows a client to have fine-grained control over the files/directories in a project: A client can retrieve from

the main repository different revisions for different files, and may choose to selectively commit files to the main repository.

Format for Storing Revisions: Storing the entire project for every new revision would imply a large storage overhead. Instead, the server stores revisions based on *delta encoding*, which takes advantage of the fact that, typically, only a small amount of data changes between revisions. This technique stores the first revision in its entirety and each subsequent revision is stored as the difference (*i.e.*, delta) relative to the previous revision. Even though this "vanilla" version of delta encoding optimizes storage on the server, retrieving a revision is not efficient: in order to obtain revision n, the server has to start from revision 1 and apply $n-1$ deltas. Instead, SVN uses *skip delta encoding*, which is a type of delta encoding further optimized towards reducing the cost of retrieving a version. A new version of a file is stored as a difference from a previous file version. However, the difference may not be relative to the immediate previous version. Skip delta encoding ensures that revision n can be obtained by computing at most $log(n)$ deltas.

When revision i is committed, the difference (*i.e.*, skip delta) is computed against a base revision which is computed as follows: in the binary representation of i, flip the rightmost bit whose value is 1. For instance, to store revision 54, the binary representation is 110110, and flipping the last "1" bit, we get 110100, which is 52. Thus, 52 is picked as the base revision for revision 54.

Notation: The VCS (main) repository contains i revisions, and we assume without loss of generality that for every file F there are i revisions which are stored as $F_0, \Delta_1, \Delta_2, \ldots, \Delta_{i-1}$. F_0 is the initial version of the file, and the $i-1$ delta files are based on skip delta encoding.

We use F_i to denote revision i of the file. We use $F_{skip(i)}$ to denote the skip version for F_i (*i.e.*, the base revision relative to which Δ_i is computed). We write

PROTOCOL: Commit

1: **for** (each file F in the commit changeset) **do**
2: $C \rightarrow S : \delta$ // Client computes and sends δ, which is the difference between F_{i-1} and F_i, such that $F_i = F_{i-1} + \delta$
3: S computes F_{i-1} based on the data in the repository (*i.e.*, start from F_0 and apply skip deltas)
4: S computes $F_i = F_{i-1} + \delta$
5: S computes $F_{skip(i)}$ based on the data in the repository (*i.e.*, start from F_0 and apply skip deltas)
6: S computes Δ_i such that $F_i = F_{skip(i)} + \Delta_i$ and stores Δ_i in the repository
7: **end for**

$F_i = F_j + \delta$ to denote that F_i is obtained by applying δ to F_j. Also, we use $C \rightarrow S : M$ to denote that client C sends a message M to the server S.

Commit protocol: The client C's local working copy contains changes made over a base revision that was previously retrieved from the server S. We refer to the changes that the client wants to commit as the *commit changeset*. Note that changes can only be committed for files for which the client has the latest revision from the server (*i.e.*, i-1). Otherwise, the client will be prompted to first retrieve the latest revision for all the files in the changeset. After C commits the changes, the latest revision at S will become i. After executing the steps described in the **Commit** protocol, the server sends the revision number i to the client, and the client sets i as the revision number for all the files in the commit changeset.

Update Protocol: The client wants to retrieve revision i for a set of files in the repository, referred to as the *update set*. After finalizing the update, the client sets i as the revision number for all the files in the update set.

2.2.2 Merkle Hash Trees

A Merkle Hash Tree (MHT) [32] is an authenticated data structure used to prove set membership efficiently. An MHT follows a tree data structure, in which every leaf node is a hash of data associated with that leaf. The nodes are concatenated

PROTOCOL: Update

1: $C \rightarrow S : i$ // C informs S that it wants to retrieve revision i
2: **for** (each file F in the update set) **do**
3: $C \rightarrow S : j$ // C sends to S the revision number for F in its local repository
4: S computes F_j and F_i based on the data in the repository (*i.e.*, start from F_0 and apply skip deltas)
5: S computes δ such that $F_i = F_j + \delta$
6: $S \rightarrow C : \delta$
7: C computes F_i as $F_i = F_j + \delta$ and stores F_i in its local repository
8: **end for**

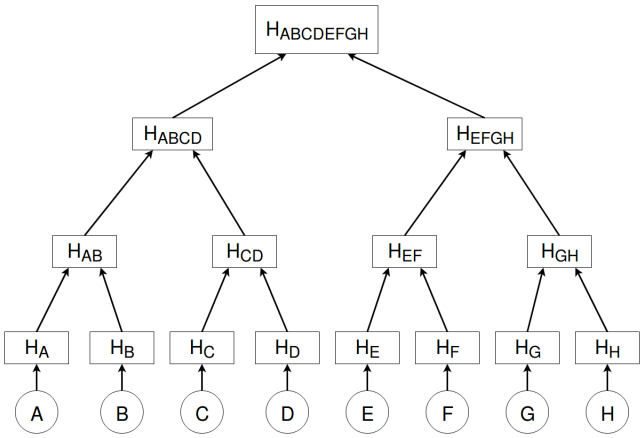


Figure 2.2 A standard Merkle Hash Tree. The set of elements consists of A, B, C, D, E, F, G, H. We use the following notation: $H_A = h(A)$, and $H_{AB} = h(H_A||H_B)$, and $H_{ABCDEFGH} = h(H_{ABCD}||H_{EFGH})$, where h is a collision-resistant hash function and "||" denotes concatenation of strings.

and hashed using a collision-resistant hash function to create a parent node, until the root of the tree is reached. Typically, a standard MHT is a full binary tree, as illustrated in Figure 2.2. Given the MHT for a set of elements, one can prove efficiently that an element belongs to this set, based on a proof that contains the root node (authenticated using a digital signature) and the siblings of all the nodes on the path between the node for the to be verified and the root node.

In this chapter, we work with sets of files and directories. As a result, we use non-standard MHTs, which are different than standard MHTs in two aspects: 1) the

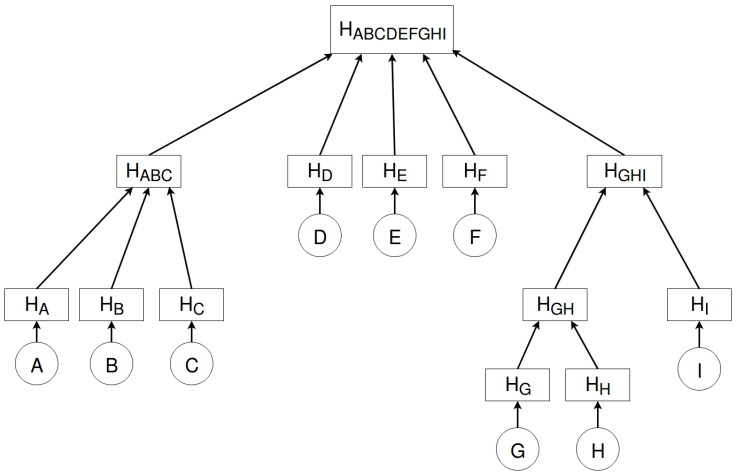


Figure 2.3 A non-standard Merkle Hash Tree.

tree is not necessarily binary (*i.e.*, internal nodes have branching factors larger than two), and 2) the tree may not be full, with leaf nodes having different depths. An internal node is obtained by hashing a concatenation of its children nodes, ordered lexicographically. This ensures that for a given repository, a unique MHT is obtained. Figure 2.3 shows an MHT in which leaves have different depths and internal nodes have different branching factors. The proof of membership for leaf node G consists of $\{H_H, H_I, H_{ABC}, H_D, H_E, H_F\}$. We will use MHTs to provide proof that a file or a set of files and directories belongs to the repository in a particular revision.

2.3 Can Git Commit Signing be Used?

In this section, we review the commit signing mechanism used in Git [10] and then identify several fundamental differences between centralized and distributed VCS-es that prevent us from using the same solution used to sign commits in Git. Git is a popular decentralized VCS, which stores the contents of the repository in form of objects [33]. Three different types of objects are used. The files in repository are represented by blob objects. A *blob object* contains the compressed content of the

(a) File system repository structure. **(b)** Git objects repository structure.

Figure 2.4 Git object hierarchy for a repository.

corresponding file and the name of the object is the hash of the file's contents. A *tree object* is used to represent a directory in the repository; it contains a list of hashes of the blob and tree objects corresponding to files and sub-directories in that directory, respectively. When the client commits to the repository, Git creates a *commit object* that is a snapshot of the entire repository at that moment, obtained as the root of an MHT computed over the repository. That commit object contains the hash of the tree object which represents the root of the repository at the time of performing this commit. Figure 2.4 illustrates the Git object hierarchy for a simple repository.

This commit object is digitally signed by the client, thus ensuring the integrity and authenticity of the repository.

We have identified several fundamental differences between Git and SVN in their workflow, functionality, and architecture. These differences make it challenging to apply the same commit signing solution used in Git to centralized VCS-es such as SVN.

Non-interactive vs. interactive commits: One important difference is that Git allows clients to perform commits without interacting with the server that hosts

the main repository, whereas in SVN clients must interact with the server. A few architectural and functional differences dictate this behavior:

- *Working with a subset of the repository:* Git relies on a distributed model, in which the entire repository (*i.e.*, all files and directories) for a given revision is mirrored on the client side. [1]. As opposed to that, SVN uses a centralized model, in which clients store locally a single revision, but have the ability to retrieve only a portion of a remote repository for that revision (*i.e.,* they can retrieve only one directory, or a subset of all the directories). This feature can be useful for very large repositories, when the client only wants to work on a small subset of the repository.

  In such cases, SVN clients do not have a global view of the entire repository and cannot use a Git-like strategy for commit signatures, which requires information about the entire repository. Instead, SVN clients must rely on the server to get a global view of the repository which raises security concerns if the server is not trustworthy and may also incur a significant amount of data transfer over the network.

- *Commit identifier:* SVN and Git use fundamentally different methods to identify a commit. Git uses a unique identifier that is computed by the client solely based on the data in that revision. This identifier is the hash of the commit object, and can be computed by the client based on the data in its local working copy, and without the involvement of the server that hosts the main remote repository. However, in SVN, the revision identifier is an integer which is chosen by the server, and which does not depend on the data in that revision. To perform a commit, the client sends the changes to the server, who then decides the revision number and sends it back to the client. Thus, a Git-like commit signature mechanism cannot be used in SVN, because clients do not have the ability to decide independently the revision identifier. This raises security concerns when the server is not trustworthy.

Working with mutually exclusive sets of files: SVN provides clients with the ability to perform commits on mutually exclusive sets of files without having to update their local working copies. For example, client A modifies a file $F1$ in directory $D1$ and client B modifies a file in another directory $D2$ of the same repository. When A wants to commit additional changes to $F1$, A does not have to update its local copy with the changes made by B. Git clients do not have this ability, as they need

[1]By default, a Git client clones locally *all* the repository revisions, but can be configured to retrieve only the last n revisions with `git clone --depth n`.

the most up-to-date version for the entire repository before pushing commits to the main repository (*i.e.*, they need retrieve all changes made anywhere in the repository before pushing new changes). This ensures that a Git client has updated metadata about the entire repository before pushing new changes. As opposed to that, SVN clients may not have the most up-to-date information for some of the files. Thus, SVN clients cannot generate or verify commit signatures in the same way as Git does, and may be tricked into signing incorrect data.

Repository Structure: SVN stores revisions of a file based on the skip delta encoding mechanism, in which a revision is stored as the difference from a previous revision. Thus, to obtain a revision for a file, the server has to start from the first revision and apply a series of deltas. On the other hand, Git stores the entire content for all versions of all files. This difference in repository structure complicates the SVN client's ability to compute and verify commit signatures. For example, a naive solution in which the client signs only the delta difference between revisions may be inefficient and insecure.

2.4 Adversarial Model and Security Guarantees

We assume that the server hosting the central repository is not trusted to preserve the integrity of the repository. For example, it may tamper with the repository in order to remove code (*e.g.*, a security patch) or to introduce malicious code (*e.g.*, a backdoor). This captures a setting in which the server is either compromised or is malicious. It also captures a setting in which the VCS server relies on the services of a third party for storing the repository, such as a cloud storage provider which may itself be malicious or may be victim of a compromise. Existing centralized VCS-es offer no protection against such attacks.

In addition to tampering with data at rest (*i.e.*, the repository), a compromised or malicious server may choose to not follow correctly the VCS protocols, as long

as such actions will not incriminate the server. For example, since commit is an interactive protocol, the server may present incorrect information to clients during a commit, which may trick clients into committing incorrect data.

When a mechanism such as commit signing is available, we assume that clients are trusted to sign their commits. In this case, we also assume that attackers cannot get hold of client cryptographic keys. The integrity of commits that are not signed cannot be guaranteed.

A malicious VCS server may execute a fork attack [34], in which some clients' actions are hidden from other clients. As a result, different clients may get different and inconsistent views of the repository. To defend against fork attacks, prior work has proposed to achieve a property called fork consistency [35], which ensures that if a server executes a fork attack, then the views seen by two inconsistent clients can never again converge. In a variation of this attack, called reconcilable fork attack [36], the server can cause clients to have inconsistent views of the repository for a finite period of time, after which the views are reconciled. To defend against such attacks, prior work has proposed to maintain at the server a cryptographically signed log that captures each history of operations performed by all clients [36]. Our focus in this chapter is not to protect against fork attacks, but to enable commit signature functionality for centralized VCS-es.

Finally, to protect the client-server communication against external adversaries, we assume that this communication occurs over secure channels, *e.g.*, the communication is secured using SSL/TLS. However, we note that even such standard mechanisms to secure communication are not always effective, and MiTM attacks have been known to occur, such as government attacks against GitHub [37, 38, 39, 40], or due to protocol flaws [41, 42, 43] and protocol misconfiguration [44, 45]. Although defending against such cases may be challenging in general, the solution we present

based on client-side commit signatures improves the security stance against such attacks as well.

2.4.1 Attacks

When the VCS employs no mechanisms to ensure repository integrity, the data in the repository is subject to a wide range of attacks as attackers can tamper with data in an arbitrary fashion. In this section, we describe a few concrete attacks that violate the integrity and authenticity of the data in the repository. This list is not meant to be comprehensive, but to suggest general requirements which are desirable for any defense that seeks to protect the integrity of the repository.

Tampering Attack. The attacker can tamper with the repository data in an arbitrary fashion, including modifying file contents, adding a file to a revision, or deleting a file from a revision. Such actions may lead to serious security integrity violations, such as the removal of a security patch or the insertion of a backdoor, which can have disastrous consequences. In SVN, there is no mechanism to detect this kind of tampering. A defense should protect against direct modification of the files.

An attacker may also try to delete a historical revision entirely, for example to hide past activity. A defense should link together consecutive revisions, such that any tampering with the sequence of revision is detected.

Impersonation Attack. The attacker can tamper with the author field of a committed revision. This will make it look like developers committed code they never actually did, which can potentially damage their reputation. Thus, a defense should protect the author field from tampering.

Mix and Match Attack. A revision reflects the state of the repository at the moment when the revision is committed. That is, the revision refers to the version of the files and directories at the moment when the commit is performed. However,

the various versions of files in the repository are not securely bound to the revision they belong to. When the server is asked to deliver a particular revision, it can send versions of the files that belong to different revisions. For example, consider a client that retrieves a subset of the repository for revision $i-1$, makes changes to it, and commits revision i. Unbeknownst to the client, the server can replace files in a directory that is not in the subset checked out by the client with old versions (*i.e.*, from before revision $i-1$), which contain a security vulnerability. As another example, consider a revision i that introduces a new feature and fixes a security vulnerability over revision $i-1$. When a new client retrieves revision i, the server provides some of the files from revision $i-1$, which still contain the vulnerability. In this attack, the server does not tamper with any file contents, but simply provides a mix of file versions from different revisions. This precludes solutions that only ensure integrity of individual files in a revision. A defense should securely bind together the files versions that belong to a revision, and should also bind them to the revision identifier.

2.4.2 Security Guarantees

Considering this adversarial model, we seek to provide the following security guarantees:

SG1: Ensure accurate commits. Commits performed by clients should be accurately reflected in the repository (*i.e.*, as if the server followed the commit protocol faithfully). After each commit, the repository should be in a state that reflects the client's actions. This allows detection of attacks in which the server does not follow the protocol and provides incorrect information to clients during a commit.

SG2: Integrity and authenticity of committed data. An attacker should not be able to modify data that has been committed to the repository without being detected. This ensures the integrity and authenticity of both individual commits and the sequence of commits. This also ensures accurate updates,

i.e., an attacker is not able to present incorrect information to clients that are retrieving data from the repository without being detected.

SG3: Non-repudiation of committed data. Clients that performed a commit operation should not be able to deny having performed that commit.

2.5 Commit Signatures for Centralized VCS-es

We have previously described the behavior of a centralized VCS system in a benign setting, in which the VCS server is fully trusted and does not deviate from the protocol. However, in this chapter we consider an adversarial setting in which the VCS server is untrusted. We now present our design for enabling commit signatures by enhancing the standard **Commit** and **Update** protocols. The commit operation generates the commit signature using the client's private key over the committed revision. The update protocol verifies the signature over a revision using the public key of the client who committed the revision.

Notation: We use the following notation, in addition to what we defined in Section 2.2.1. $CSIG_i$ denotes the client's commit signature over revision i, and $MHTROOT_i$ denotes the root of the Merkle hash tree built on top of revision i. We use $Sign$ and $Verify$ to denote the signing and verification algorithms of a standard digital signature scheme. To simplify the notation, we will omit the keys, but $Sign$ uses the client's private key and $Verify$ uses the client's public key.

2.5.1 Secure Commit Protocol

We now present **Secure_Commit**, an enhanced version of the standard **Commit** protocol (Section 2.2.1), which incorporates commit signatures. Recall that the client has a commit changeset that contains changes on top of revision $i-1$, and the client wants to commit revision i. The first steps of **Secure_Commit** are the same as in the standard **Commit** protocol (lines 1-6).

The client needs to compute the commit signature over revision i of the entire repository. However, the client's local working copy may only contain a subset of the entire repository (*e.g.*, only the files that are part of the commit changeset). Thus, in order to compute the commit signature, the client needs additional information from the server about the files in the repository that are not in its local working copy. The server will provide this additional information in the form of a proof relative the client's changeset (line 7). We describe how this proof is computed and verified in Section 2.5.3. After receiving the new revision number, the proof, and the commit signature and revision information for revision $i-1$ (line 8), the client verifies the validity of the proof (line 9). The client then uses this proof and the files in the changeset to compute the root of the MHT over revision i of the repository (line 10). Finally, the client computes the commit signature over revision i as a digital signature over the root of the MHT and the revision information (which includes the current revision number i, the previous revision number $i-1$, and the client's ID as the author of the commit) (line 12). Upon receiving the commit signature (line 13), the server recomputes the MHT for revision i and stores it together with the client's commit signature and revision information (lines 14-15).

2.5.2 Secure Update Protocol

The client wants to retrieve revision i for a set of files in the repository, referred to as the *update set*. **Secure_Update** shares the first steps with the standard **Update** protocol, in which the client retrieves the difference (*i.e.*, deltas) between revision j and revision i for the files in the update set (lines 1-7). To allow the client to check the authenticity of the deltas, the server computes a proof for the MHT build on top of revision i, relative to the client's update set (line 8). The server sends this proof to the client, together with the commit signature and revision information for revision i

PROTOCOL: Secure_Commit

1: **for** (each file F in the commit changeset) **do**
2: $C \rightarrow S : \delta$ // Client computes and sends δ, which is the difference between F_{i-1} and F_i, such that $F_i = F_{i-1} + \delta$)
3: S computes F_{i-1} based on the data in the repository (*i.e.*, start from F_0 and apply skip deltas)
4: S computes $F_i = F_{i-1} + \delta$
5: S computes $F_{skip(i)}$ based on the data in the repository (*i.e.*, start from F_0 and apply skip deltas)
6: S computes Δ_i such that $F_i = F_{skip(i)} + \Delta_i$ and stores Δ_i in the repository
7: **end for**
8: S computes proof P_{i-1} // S uses revision $i-1$ if the repository to compute a proof relative to the client's commit changeset
9: $S \rightarrow C : i, P_{i-1}, CSIG_{i-1}, RevInfo_{i-1}$ // S sends the new revision number i, the proof for the changeset, and the commit signature and revision information for revision $i-1$
10: **if** $(Verify(CSIG_{i-1}) == invalid)$ **then** C aborts the protocol // C verifies the commit signature using $P_{i-1}, RevInfo_{i-1}$ and revision $i-1$ of the files in the commit changeset
11: **end if**
12: C computes the $MHTROOT_i$ using P_{i-1} and revision i of the files in the commit changeset
13: C sets $RevInfo_i = i, i-1, ID_{client}$
14: C computes $CSIG_i = Sign(MHTROOT_i, RevInfo_i)$
15: $C \rightarrow S : CSIG_i, RevInfo_i$
16: S computes the MHT for revision i using the MHT for revision $i-1$ and the client's changeset
17: S stores $CSIG_{i-1}, RevInfo_i$ and the MHT for revision i

(line 9). The client then verifies this proof (line 10). After finalizing the update, the client sets i as the revision number for all the files in the update set.

2.5.3 MHT-based Proofs

As described in the previous sections, the commit signature $CSIG_i = Sign(MHTROOT_i, RevInfo_i)$ binds together via a digital signature the root of a Merkle Hash Tree (MHT) with the revision information, both computed over revision i. In the **Secure_Commit** and **Secure_Update** protocols, the client relies on an MHT-based proof from the server to verify the validity of information provided by the server that

PROTOCOL: Secure_Update

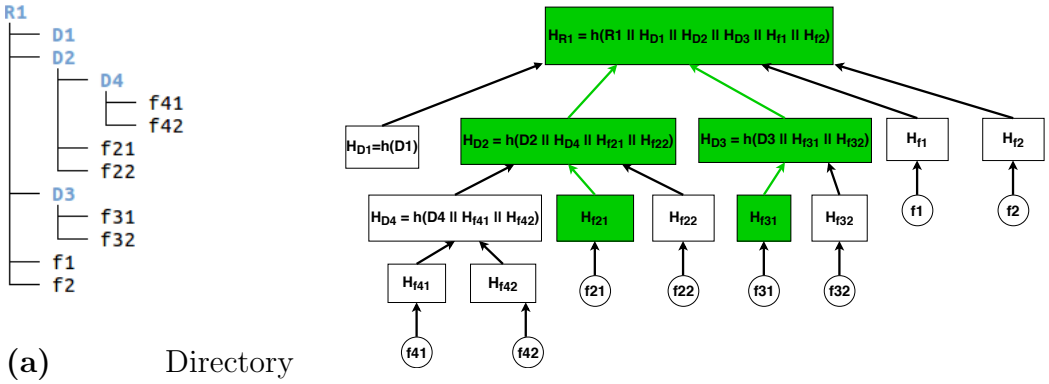
1: $C \rightarrow S : i$ // C informs S that it wants to retrieve revision i
2: **for** (each file F in the update set) **do**
3: $C \rightarrow S : j$ // C sends to S the revision number for F in its local repository
4: S computes F_j and F_i based on the data in the repository (*i.e.*, start from F_0 and apply skip deltas)
5: S computes δ such that $F_i = F_j + \delta$
6: $S \rightarrow C : \delta$
7: C computes F_i as $F_i = F_j + \delta$
8: **end for**
9: S computes proof P_i // S uses revision i of the repository to compute proof P_i relative to the client's update set
10: $S \rightarrow C : P_i, CSIG_i, RevInfo_i$ // S sends the proof for the update set, and the commit signature and revision information for revision i
11: **if** $(Verify(CSIG_i) == invalid)$ **then** C aborts the protocol // C verifies the commit signature using $P_i, RevInfo_i$ and revision i of the files in the update set
12: **end if**
13: **for** (each file F in the update set) **do**
14: C stores F_i in its local repository
15: **end for**

is not present in the client's local repository. This covers scenarios in which the client works locally with only a portion of the repository. We now describe how such a proof can be computed and verified.

MHT for a repository. To compute the commit signature, an MHT is built over the a revision of the repository. The leaves of the MHT consist of hashes of files. These hashes are concatenated and hashed to get the hash of the parent directory. This process continues recursively until we obtain the root of the MHT. Figure 2.5 shows the directory structure and the corresponding MHT for a revision of repository R1.

MHT-based proofs. The client relies on a proof from the server to verify the validity of information received from the server relative to a set of files that it stores locally (*i.e.*, the commit changeset for a commit, or the update set for an update).

The proof of membership for an element contains the siblings of all the nodes on the path between the node to be verified and the root node. For example,



(a) Directory structure of repository R1.

(b) MHT for repo R1. Also shown is the Steiner tree for the set of files {f21, f31}.

Figure 2.5 MHT for a revision of repository R1.

consider the MHT for the repository R1 as shown in Figure 2.5b. The proof for node H_{f31} is $\{H_{f32}, H_{D1}, H_{D2}, H_{f1}, H_{f2}\}$, whereas the proof for node H_{f21} is $\{H_{D4}, H_{f22}, H_{D1}, H_{D3}, H_{f1}, H_{f2}\}$ We can see that nodes H_{D1}, H_{f1}, and H_{f2} are repeated in the proofs of these two nodes. Thus, when computing a proof of verification for multiple nodes in the MHT, many of the nodes at higher levels of the tree will be common to all the nodes and will be sent multiple times.

To avoid unnecessary duplication and to reduce the data sent from server to client, we follow an approach based on a Steiner tree to compute the proof on the server side. For a given tree and a subset of leaves of that tree, the Steiner tree induced by the set of leaves is defined as the minimal subtree of the tree that connects all the leaves in the subset. This Steiner tree is unique for a given tree and a subset of leaves. The proof for a set of nodes consist of the nodes that "hang off" the Steiner tree induced by the set of nodes (*i.e.*, siblings of nodes in the Steiner tree). Using the same example as earlier, the Steiner tree for the set of

nodes $\{H_{f21}, H_{f31}\}$ if shown in Figure 2.5b using solid-filled nodes. Thus, the proof is $\{H_{D4}, H_{f22}, H_{f32}, H_{D1}, H_{f1}, H_{f2}\}$.

2.6 Security Analysis (Sketch)

During a commit, the client needs to compute the MHT for revision i. A client that stores locally only a portion of the repository for revision $i-1$ relies on the server to get information about other parts of the repository that are not stored locally. The server provides this information as a proof relative to the client's commit changeset, and the client verifies the authenticity of this information by recomputing the root of the MHT for revision $i-1$ and validating it against the commit signature for revision $i-1$. This prevents the server from providing incorrect information, and thus our commit signing mechanism achieves **security guarantee SG1**.

For every commit, the commit signature covers the entire revision data and the additional revision information, which includes the revision number for this commit, the revision number for the previous commit, and the commit author's ID. This prevents an attacker from tampering with any individual commit and not be detected. Since the signature covers the revision data, files cannot be tampered with, for example to add, delete, or change file content. Since the signature covers the commit author's ID, impersonation attacks are prevented. Since the signature contains the revision number of the previous commit, this creates a chain of commits which prevents an attacker from altering the sequence of commits by removing, adding, or reordering entire commits. The commit signature binds together the files and directories in a revision. As a result, an attacker cannot claim that a file from one revision belongs to a different revision.

During an update, the client gets a proof from the server related to information that the client does not have locally, which allows the client to compute the MHT for revision i, and verify it agains the commit signature for revision i. This ensures

that the server cannot deviate from protocol by providing incorrect information about the update set requested by the client. Thus, we conclude that our commit signing mechanism achieves **security guarantee SG2**.

Security guarantee SG3 results directly from the non-repudiation guarantee offered by the digital signature used to obtain the commit signature.

2.7 Implementation and Experimental Evaluation

2.7.1 Implementation

We implemented SSVN by adding approximately 2,500 lines of C code on top version 1.9.2 of the SVN codebase. For cryptographic functionality, we used the following primitives from the OpenSSL version 1.0.2g: RSA with 2048-bit keys for digital signatures, and SHA1 for hashing.

Implementation Overview: SSVN provides integrity, authentication and non-repudiation for the repositories stored on the server. We have modified the source code for both SVN client and SVN server. The server now stores additional metadata for each revision, as follows:

– the nodes of a Merkle Hash Tree (MHT) computed over that revision; this allows the server to efficiently compute proofs for the client during the commit and update protocols;

– the commit signature for that revision generated by the client.

For the SVN client, we made changes to the following SVN commands:

– svn commit: commits the changes to the repository. The corresponding new command is "ssvn commit".

– svn update: fetches a revision from the server. The corresponding new command is "ssvn update".

For the SVN server, we made changes to "svnserve". The corresponding new command in SSVN is "ssvnserve". In addition to modifying existing SVN

commands, we added the "svn keys" command for the SVN client to generate a pair of public/private pair of keys that can be used for commit signatures.

When committing a revision, we added functionality so that the server computes and sends to the client a proof relative to the client's changeset. The client uses this proof to verify the signature over the previous revision. The client then uses parts of this proof to compute the root of the MHT and the commit signature for the new revision. The commit signature is sent to the server, where it is stored as metadata. The server computes the MHT over the new revision using the MHT over the previous revision and the client's changeset.

When the client requests a revision, the server sends the actual revision data together with the commit signature for that revision. In addition, the server uses the stored MHT for that revision in order to compute and send to the client a proof relative to the client's update set. The client uses this proof and the commit signature to verify the validity of the data in the retrieved revision.

Storing the MHT at the server: When a new revision is created at the server, the server stores an MHT for that revision. We implemented two variants of storing this MHT at the server:

– store the entire MHT for each revision. This has the advantage of allowing the server to compute directly proofs based on the MHT.

– use skip-delta encoding to store the MHT incrementally as the difference from the MHT of a previous revision. We re-used the skip-delta functions in the SVN codebase, which are normally used to encode repository data incrementally. Whenever the server needs the MHT for a revision, it uses the skip-deltas to reconstruct that MHT. This approach has the advantage of reducing the storage overhead at the server, but requires the server to perform some additional computation during the interaction with clients.

Table 2.1 Statistics for the Selected Repositories (as of March 2018). The number of Files and the Average File Size are Based on the Latest Revision in the Repository

	FileZilla	SVN	GCC
Number of revisions	8,738	1,826,802	258,555
Number of Files	1,454	2,207	79,552
Average File Size	21KB	18KB	6KB
Repository size (all revisions)	29.2MB	43.9MB	492.7MB

2.7.2 Experimental Setup

We ran experiments with both SVN server and SVN clients running on the same machine, an Intel Core i7 system with 4 cores (each running at 2.90 GHz), 16GB RAM, and a 500GB hard disk with ext4 file system. The system runs Ubuntu 16.04 LTS, kernel version 4.10.14-041014-generic, and the OpenSSL 1.0.2g library.

Repository selection. For the experimental evaluation, we wanted to cover a diverse set of repositories with regard to the number of revisions, number of files, and average file size. Thus, we have chosen three representative public SVN repositories: FileZilla [46], SVN [11], and GCC [47]. Statistics for these repositories are provided in Table 2.1.

Overview of experiments. We have evaluated the end-to-end delay, and the communication and storage overhead associated with the commit and update operations for both SSVN and SVN. We average the overhead over the first 100 revisions of the three selected repositories (labeled FileZilla, SVN, and GCC1). GCC is a large size repository, with over 250K revisions and close to 80K files. Since for GCC the difference between the first 100 revisions and the last 100 revisions is considerable in the size of the repository, we also included in our experiments an average of the overhead over the last 100 revisions of GCC (labeled GCC2).

Table 2.2 Commit Time for One Revision (in seconds)

	FileZilla	SVN	GCC1	GCC2
SVN	0.183	0.300	0.385	7.342
SSVN	0.248	0.336	0.459	8.217

All the data points in the experimental evaluation section are averaged over three independent runs.

2.7.3 Experimental Evaluation for Commit Operations

End-to-end delay. To measure the end-to-end delay for a commit operation, we measured the time needed for running the shell commands "svn commit" and "ssvn commit" to commit a revision.

The experimental results are shown in Table 2.2. We can see that, compared to SVN, SSVN increases the end-to-end delay between 12% (for SVN) and 35% (for FileZilla). The overhead is smaller for the SVN repository because the changeset in each commit is small, and thus the corresponding change in the MHT metadata is also small. Even though 35% is a large relative increase for the FileZilla repository, we note that the increase is only 0.06 seconds per commit. For the GCC repository, the overhead decreases from 20% to 12% as we look at the first 100 revisions compared to the last 100 revisions. This is because the changeset in a commit represents a smaller percentage as the size of the files in the GCC codebase increases. In absolute terms, the increase for GCC remains less than 1 second.

Communication overhead. To measure the communication overhead for a commit operation, we observed that both SVN and SSVN clients rely on two write functions *writebuf_output* and *svn_ra_svn_writebuf_output* to send data, and on two read functions *readbuf_input* and *svn_ra_svn_readbuf_input* to receive data. Thus, for

Table 2.3 Server Storage for One Revision (in MBs)

	FileZilla	SVN	GCC1	GCC2
SVN	4.504	0.514	4.263	20.346
SSVN	4.610	0.682	4.415	23.563

Table 2.4 Network Communication from Client to Server for Committing One Revision (in KBs)

	FileZilla	SVN	GCC1	GCC2
SVN	35.565	46.672	4.676	20.347
SSVN	35.825	46.934	4.933	20.605

each commit operation, we accumulate the data sent in the write functions, which are the total communication from the client to the server. Similarly, we accumulate the data received in the read functions, which are the total communication from the server to the client.

Table 2.4 shows that SSVN adds about 256 bytes to the communication from client to server, which matches the size of the commit signature that is sent by the client with committing a revision. Table 2.5 shows that SSVN adds between 0.27KB to 0.8KB of communication overhead from server to client. This overhead is caused by

Table 2.5 Network Communication from Server to Client for Committing One Revision (in KBs)

	FileZilla	SVN	GCC1	GCC2
SVN	0.865	1.095	0.539	2.476
SSVN	1.137	1.432	0.962	3.275

the verification metadata sent by server which the client uses to verify the signature over previous commit and to generate the signature for this commit.

Storage overhead. We measured the storage overhead incurred by SSVN on the server as the difference between the size of the server repository after every commit operation for SVN and SSVN. There is no storage overhead on the client side as the client does not store any additional data in SSVN. Table 2.3 shows that SSVN adds between 0.1MB - 0.16MB per commit over SVN for FileZilla, SVN, and GCC1. This reflects the fact that the server stores one MHT per revision and the size of the MHT is proportional to the number of files in the repository. We also see the storage overhead increases significantly between GCC1 and GCC2. The reason for the increase is that the number of files in the GCC repository increases significantly from revision 1 (about 3,000 files) to the latest revision (close to 80,000 files). Since the MHT is proportional to the number of files, the storage overhead for recent revisions in the GCC repository increases to about 3MB.

2.7.4 Experimental Evaluation for Update Operations

End-to-end delay. To measure the end-to-end delay for an update operation, we measured the time needed to run the shell commands "svn update -r i" (for regular SVN) and "ssvn update -r i" (for SSVN) to retrieve revision i by updating revision $i-1$.

The experimental results are shown in Table 2.6. We observe that the time needed retrieve a revision in SSVN increases between 11% and 41% compared to regular SVN. Even though 41% looks high, note that the increase is quite modest as an absolute value, at 0.03 seconds. Even for GCC2, the maximum increase remains modest, at 0.638 seconds. This increase is caused by the time needed to generate the proof on the server side, to send the proof to the client, and to verify the proof on the client side.

Table 2.6 Update Time for One Revision (in seconds)

	FileZilla	SVN	GCC1	GCC2
SVN	0.072	0.098	0.150	3.215
SSVN	0.098	0.109	0.182	3.853

Table 2.7 Comparison Between SSVN and SSVN-skip for a Large Repository (the last 100 revisions in GCC)

	SVN	SSVN	SSVN-skip
commit time (seconds)	7.342	8.217	9.143
update time (seconds)	3.215	3.853	4.977
server storage (MBs)	20.346	23.563	21.162

Communication overhead. To measure the network communication overhead, we followed the same procedure as for commit operations. The experimental results are shown in Tables 2.8 and 2.9. SSVN adds between 0.24KB - 0.66KB to the communication from the server to the client. This overhead is caused by the proof that the server sends to the client, which is required on the client side to verify the commit signature for the requested revision.

Table 2.8 Network Communication from Client to Server for Updating One Revision (in KBs)

	FileZilla	SVN	GCC1	GCC2
SVN	1.243	1.328	0.953	10.235
SSVN	1.235	1.548	1.045	11.369

Table 2.9 Network Communication from Server to Client for Updating One Revision (in KBs)

	FileZilla	SVN	GCC1	GCC2
SVN	36.342	49.978	5.782	54.678
SSVN	36.745	50.225	6.245	55.346

2.7.4.1 A Storage-Computation Tradeoff for Large Repositories.

As seen previously, SSVN incurs a relatively large storage overhead (over 3 MB) for a large repository such as GCC2. We evaluate the performance of our alternative implementation in which the server stores the MHT per revision in an incremental fashion, using skip-delta encoding. Table 2.7 shows a comparison between SVN, SSVN and SSVN-skip for a large repository, more specifically for the last 100 revisions in GCC. We observe that SSVN-skip provides an advantageous tradeoff between storage and computation over SSVN: it reduces storage overhead by 75% (from 3.217 MB to 0.816 MB) at the cost of increasing the computation by 11% for a commit and by 29% for an update.

2.8 Related Work

Even though an early proposal draft for SVN changeset signing has been considered [48], it only contains a high-level description and lacks concrete details. It has not been followed by any further discussion regarding efficiency or security aspects, and it did not lead to an implementation. Furthermore, the proposal suggests to sign the actual changeset, which may lead to inefficient and insecure solutions, and does not cover features such as allowing partial repository checkout, or allowing clients to work with disjoint sets of files without having to retrieve other clients' changes.

GNU Bazaar [27] is a centralized VCS that allows to sign and verify commits [49] using GPG keys. However, although Bazaar supports features such as partial

repository checkout and working with disjoint sets of files, commit signing is not available when these features are used.

Wheeler [50] provides a comprehensive overview of security issues related to source code management (SCM) tools. This includes security requirements, threat models and suggested solutions to address the threats. In this chapter, we are concerned with similar security guarantees for commit operations, *i.e.*, integrity, authenticity and non-repudiation.

Git provides GPG-based commit signature functionality to ensure the integrity and authenticity of the repository data [51]. Gerwitz [52] gives a detailed description of Git signed commits and covers how to create and verify signed commits for a few scenarios associated with common development workflows. As we argued earlier in the chapter (Section 2.3), several fundamental architectural and functional differences prevent us from applying the same commit signing solution used in Git to centralized VCS-es such as SVN.

Chen and Curtmola [53] proposed mechanisms to ensure that all of the versions of a file are retrievable from an untrusted VCS server over time. The focus of their work is different than ours, as they are concerned with providing probabilistic long-term reliability guarantees for the data in a repository. Relevant to our work, they provide useful insights into the inner workings of VCS-es that rely on delta-based encoding.

2.9 Conclusion

In this chapter, we introduce a commit signing mechanism that substantially improves the security model for an entire class of centralized version control systems (VCS-es), which includes among others the well-known Apache SVN. As a result, we enable integrity, authenticity and non-repudiation of data committed by developers. These security guarantees would not be otherwise available for the considered VCS-es.

We are the first to consider commit signing in conjunction with supporting VCS features such as working with a subset of the repository and allowing clients to work on disjoint sets of files without having to retrieve each other's changes. This is achieved efficiently by signing a Merkle Hash Tree (MHT) computed over the entire repository, whereas the proofs about non-local data contain siblings of nodes in the Steiner tree determined by items in the commit/update changeset. This technique is of independent interest and can also be applied to distributed VCS-es like Git in case Git moved to support partial checkouts (a feature that has been considered before) or in ongoing efforts to optimize working with very large Git repositories ([54, 55]).

We implemented a prototype on top of the existing SVN codebase and evaluated its performance with a diverse set of repositories. The evaluation shows that our solution incurs a modest overhead: for medium-sized repositories we add less than 0.5KB network communication and less than 0.2 seconds end-to-end delay per commit/update; even for very large repositories, the communication overhead is under 1KB and end-to-end delay overhead remains under 1 second per commit/update. We also explored a skip-delta encoding optimization for the MHT-metadata stored on the server, which provides a significant storage overhead reduction for large repositories.

CHAPTER 3

BOOTSTRAPPING TRUST FOR SOFTWARE REPOSITORIES

Community repositories such as PyPI and NPM are immensely popular and collectively serve more than a billion packages per year. However, existing software certification mechanisms such as code signing, which seeks to provide to end users authenticity and integrity for a piece of software, are not suitable for community repositories and are not used in this context. This is very concerning, given the recent increase in the frequency and variety of attacks against community repositories.

In this chapter, we propose a different approach for certifying the validity of software projects hosted on community repositories. We design and implement a *Software Certification Service (SCS)* that receives certification requests from a project owner for a specific project and then issues a project certificate once the project owner successfully completes a protocol for proving ownership of the project. The proposed certification protocol is inspired from the highly-successful ACME protocol used by Let's Encrypt and can be be fully automated on the SCS side. However, it is fundamentally different in its attack mitigation capabilities and in how ownership is proven. It is also compatible with existing community repositories such as PyPI, RubyGems, NPM, or GitHub, without requiring any changes to these repositories. To support this claim, we show how to instantiate the proposed certification service with several practical deployments.

3.1 Introduction

Community repositories such as PyPI [56], RubyGems [57], and NPM [58] are among the most popular and accessible way of publishing and distributing open source software. Their immense popularity is illustrated by the large number of downloads: PyPI, the Python package manager, sees more than 400 million downloads

per day [59]; NPM, the Javascript package manager, more than 500K downloads a day [60]; RubyGems, the public repository of Ruby packages, has seen more than 83 billion downloads since its creation (as of October 2021). Due to their popularity, attacks against community repositories have been on the rise in the recent past [61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72]. For instance, in July 2021, a PyPI package containing a backdoor was downloaded almost 30,000 times before the breach was detected [61]. In April 2020, a supply chain attack on RubyGems used packages with names similar to popular packages to infect the end user's system [66]. Similar types of supply chain attacks have become a rising concern for users of NPM as well [67, 68, 69, 70, 71, 72]. The increase in the frequency and variety of attacks against community repositories makes it necessary to improve the overall security stance of these popular custodians of open-source software. In this chapter, we focus on a fundamental question: *How can end users retrieve an authentic version of a community repository project, as intended by the project owner?* End users can bootstrap trust in a software project by ensuring that what they retrieve is what the project owner intended. When a software project is digitally signed, this question becomes: *How can end users obtain an authentic version of the project owner's public key?*

When turning to existing mechanisms to certify software, we realized that they may not be appropriate in the context of community repositories. While code signing certificates [73, 74, 75] ostensibly provide a means to validate the identity of the software publisher, apart from a few large companies, they are rarely used in practice. This sort of certification often requires out-of-band verification and cannot be easily automated. As a result, unfortunately, the effort to required to obtain a certificate is prohibitive, making these unsuitable for all types of software projects. We elaborate more in Section 3.3 on the limitations of existing certification mechanisms, including code signing.

In this chapter, we propose a different approach for certifying the validity of software projects hosted on community repositories. To leverage the existing PKI model of trust, our goal is to provide a way to bootstrap trust using this mechanism. In the PKI model, a certification authority binds a domain owner and a domain name to a public key. The domain owner provides proof of ownership in order to get the X.509 domain certificate. Similarly, we propose a solution where the software project owner proves the ownership of the project and gets a digital certificate that binds the project owner and the project name to a public key. We design and implement a *Software Certification Service (SCS)* that receives certification requests from a project owner for a specific project and then issues a project certificate once the project owner successfully completes a procedure for proving ownership of the project. This project certificate validates a public key for the project.

Unlike in the code signing model, which seeks to establish trust in the identity of the software publisher using a cumbersome procedure, the proposed Software Certification Service relies on a certification protocol with the project publisher to establish ownership of the software. The proposed certification protocol is inspired from the highly-successful ACME protocol [76] used by Let's Encrypt [2] and can be be fully automated on the SCS side. However, it is fundamentally different in its attack mitigation capabilities (*i.e.*, compromise resiliency) and in how ownership is proven (*e.g.*, how to account for the specifics of software naming as opposed to domain names). It is also compatible with existing community repositories such as PyPI, RubyGems, NPM, or GitHub, without requiring any changes to these repositories.

In the ACME protocol, the owner of a domain proves ownership of that domain by provisioning a specific HTTP resource (*e.g.*, random token chosen by the Let's Encrypt CA) at a specific URL at that domain. In our approach, the project publisher proves ownership over a project by executing a certification protocol with the SCS, which requires the publisher to answer SCS challenges by provisioning certain HTTP

resources (*e.g.*, random tokens chosen by the SCS) at a specific location on the project's webpage. To preserve the functionality of the project description field and reduce confusion for the casual user who browses the project's webpage, the random tokens are placed at the end of the project description, using delimiters that make it clear they are not part of the actual project description. The ability to answer SCS challenges proves control over the project's repository. After successfully completing the certification protocol, the project owner gets a project certificate which binds a public key to a (project ID, project owner) tuple. The project owner then uses the corresponding private key to sign the software project for distribution to end users.

The SCS certification protocol includes safeguards to provide resiliency against an adversary who is able to control a project repository (*e.g.*, by compromising the project repository credentials). First, the certification protocol is designed to last over an extended period of time. we raise the bar to adversaries who must maintain control over a project for a prolonged period of time, which is arguably more difficult to achieve while going undetected. Second, the SCS protocol requires that the response to a challenge must be placed on the project's repository in a publicly visible way (*i.e.*, the project's webpage). This will prevent an adversary to execute the certification protocol in a stealthy manner.

Finally, we instantiate the proposed service with several practical deployments. First, we use the service to automate the certification of community repositories projects. Our deployments include several popular community repositories: PyPI, RubyGems, NPM, and GitHub. Second, we use the service to automate the delegation process in community repositories that rely on systems like TUF [3, 4, 5] to provide compromise resilience.

3.2 Background

The software certification protocol proposed in this chapter is modeled after the Automatic Certificate Management Environment (ACME) protocol [76]. This section provides an overview of the ACME protocol.

ACME is a protocol that a certificate authority (CA) and an applicant can use to automate the process of verification and HTTPS certificate issuance. Certificate issuance using ACME resembles a traditional CA's issuance process, in that a user creates an account, requests a certificate for a domain, and proves control of the domain in that certificate in order for the CA to issue the requested certificate.

The entities interacting in the ACME protocol are the ACME client (*i.e.*, the applicant for the HTTPS certificate) and the ACME server (*i.e.*, the CA who issues HTTPS certificates). To begin the process of certificate issuance, the ACME client generates a key pair whose public key will be included in the HTTPS certificate to be generated by the CA. The client proves knowledge of the corresponding private key by signing a CSR (certificate signing request).

The client then engages in a protocol with the ACME server to prove control over the requested domain. For this, the client needs to complete a challenge issued by the ACME server. The client can choose from two types of challenges. A *DNS challenge* requires the client to provision a DNS record under the requested domain name. An *HTTP challenge* requires the client to provision an HTTP resource under the domain's URI.

Figures 3.1 and 3.2 illustrate the process of getting a certificate for the domain *example.com*. In Figure 3.1, the client asks for a certificate for *example.com* and the ACME server responds with an HTTP challenge that requires the client to provision a file at a specified location under the domain name. In Figure 3.2, the client completes the challenge and informs the server. The server then checks if the client has indeed completed the challenge and informs the client that validation is complete.

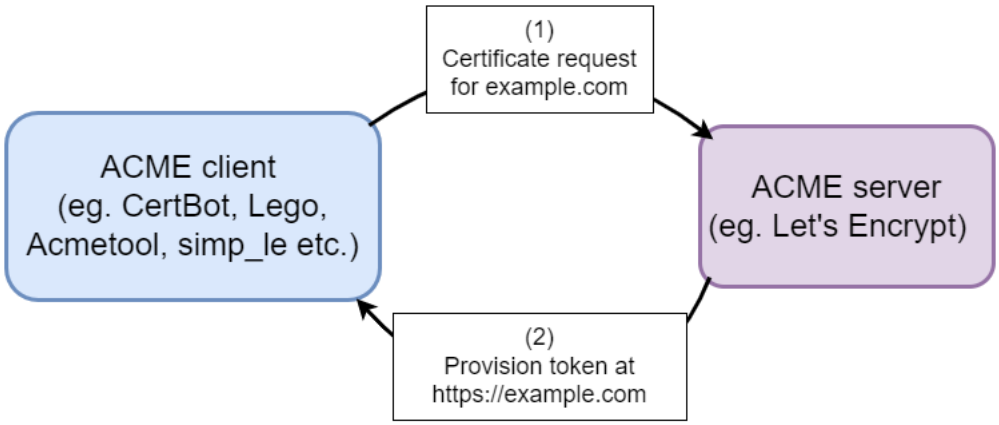


Figure 3.1 The CA asks the client to prove control over domain name *example.com*.

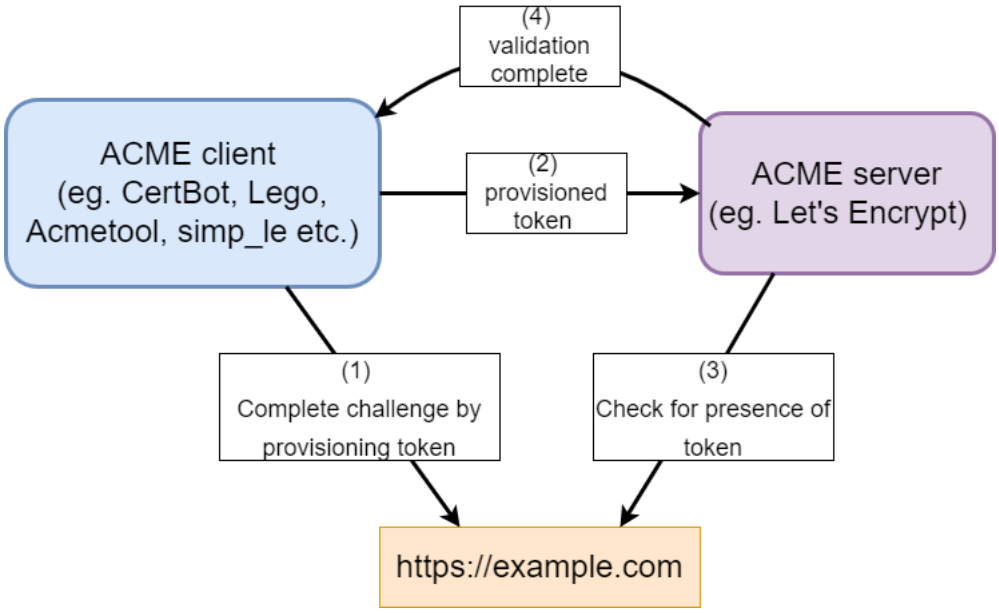


Figure 3.2 The CA verifies that the client has proven control over domain name *example.com*.

Once the validation is successful, the client sends a certificate signing request (CSR) just like in the traditional certificate issuance process. On receiving the request, the CA issues the certificate.

The ACME protocol looks very similar to the traditional certificate issuance protocols. However, the major difference lies in the step where the client proves control over the domain. For a traditional CA, this step requires human intervention. Instead, ACME automates the process of proving control (based on DNS or HTTP challenges) and it also automates the sever's validation process for these challenges.

The Let's Encrypt CA. Let's Encrypt is a free, automated, and open certificate authority. The objective of Let's Encrypt is to make it possible to set up an HTTPS server and have it automatically obtain a browser-trusted certificate, without any human intervention. Towards this goal, Let's Encrypt uses the ACME protocol to issue certificates. Let's Encrypt makes the process of certificate issuance easy and affordable by making it automated and free.

Since its debut in September 2015, Let's Encrypt has grown rapidly to become the largest CA on the web. As of November 2021, Let's Encrypt serves 275 million websites with over 219 million active certificates [77]. This has contributed to a significant increase in the HTTPS traffic on the web. Since 2013, HTTPS page loads have grown from 25% to 82% globally and over 93% in the United States [77].

3.3 Existing Mechanisms

This section gives an overview of existing software certification mechanisms.

3.3.1 Code Signing

The code signing model mirrors the PKI model used to issue domain validation TLS X.509 certificates. A software publisher applies for a publisher certificate with a Certificate Authority (CA) and proves its identity in the process. Having verified the publisher's identity, the CA issues a *code signing certificate* which binds the identity of the software publisher to a public key. The publisher then signs the software using the private key corresponding to the public key in the certificate. Finally, the user

downloads the signed software, verifies the signature, and validates the publisher's certificate.

Code signing provides the following two guarantees: (1) Validation of the software publisher, *i.e.*, the software comes from a known publisher, and (2) Software integrity (*i.e.*, it has not been modified since it was signed and released by the publisher). The code signing certificate that accompanies the software provides a guarantee that certain checks were done by the CA about the identity of the publisher. As such, it fits best scenarios in which end users need to establish the trustworthiness of the publisher.

Unfortunately, to have its identity verified by the CA, the software publisher needs to go through a very cumbersome process. In addition to verifying that the publisher controls the domain name(s) listed on certificate, the CA need to verify the legal, physical and operational existence of the publisher's business before issuing the certificate. This requires the publisher to provide proof about the legal creation, existence, or recognition of the business (*e.g.*, record in a business registry database and in a trusted online directory), proof of financial standing with a regulated financial institution (*e.g.*, a D-U-N-S number, or a letter from a certified public accountant,), and signed copies of various legal agreements. The publisher must be available to answer phone calls: an initial call to validate the company's phone number and a final verification call to validate all of the business and certificate-related details. The CA must also verify the name, title, authority and signature of the person(s) involved in requesting the certificate and agreeing to the terms and conditions.

Code signing certificates are mainly targeted for businesses, with a few CAs issuing them to individuals, in which case the applicant needs to supply a copy of government issues photo ID (*e.g.*, driver's license or passport), a high-resolution photo of the applicant holding their ID next to their face, and a copy of a recent major utility bill or bank statement in the applicant's name.

Given this manual and lengthy validation process, code signing certification cannot be automated and imposes large operational costs for the CA, which needs to hire staff and invest in an infrastructure to ensure a rigorous validation process. The current code signing model may not be suitable for all types of software, as it might be difficult for small businesses, start-ups, independent developers and freelancers to afford a code signing certificate (which few users will validate) that incurs significant costs.

Another direct consequence of the cumbersome and intrusive certification process is the low adoption rate for code signing certificates. This model for certifying software is endorsed by several large software publishers such as Microsoft (for the Windows ecosystem and MS Office objects), Apple (for software developed using Xcode), and Adobe (for Adobe Air applications). Besides these limited use cases inside closed ecosystems, code signing remains largely unused for the large majority of software, including open source software.

3.3.2 Package Signatures

Some community repositories allow their packages to be cryptographically signed with a private key so that end users can verify the packages with a public key. There are generally two types of package signing, discussed next.

Signed-by-repository: In community repositories such as as npm, the repository signs uploaded packages with a repository private key. The corresponding public key is publicized on Keybase [78] and is used by end users to verify downloaded packages. The repository private key is kept online to ensure that new packages can be signed as soon as possible. This results in a coarse-grained security guarantee. A compromise of the repository invalidates the security of all its packages. If, on the other hand, the repository private key remains secure, a package signature guarantees that the package uploaded to the repository is the package that an end user downloads. This

in itself does not account for the possibility that an individual project's credentials were compromised (even for a brief amount of time) and a malicious version of a package was uploaded to the community repository.

Signed-by-author: In community repositories such as RubyGems, a package is signed by its author before being uploaded. The private key used for signing is kept offline. In turn, end users verify the end-to-end authenticity of the downloaded packages based on the corresponding public key. The problem with this model is that end users must discover the correct key for an author by using out-of-band channels, a manual process that is vulnerable to fake key distribution attacks. An alternative method to establish the authenticity of a public key is to use a PGP decentralized "web of trust", in which authors vouch for each other's GPG keys.

3.3.3 Binary Transparency

This model was considered by Mozilla to distribute the Firefox browser [79]. When a binary release of Firefox is created, a description of that release is logged to a publicly verifiable log. The approach taken is to layer this Binary Transparency system on top of the existing Certificate Transparency (CT) system, by computing a hash value that summarizes the release (this is the head of a Merkle hash tree computed over the digests of all binaries in the release), putting that value in a certificate, and logging the certificate in a CT log.

The certificate is a DV (domain validation) X.509 certificate that is obtained by Mozilla from the Let's Encrypt CA for a domain of the form `<Hash-of-Merkle-tree>.<release-number>.fx-trans.net`.

Then, a third party can verify that a binary was logged by verifying that the binary is included in the Merkle tree with the tree head included in the certificate, and then verifying that the certificate was logged in the CT log. This ensures that

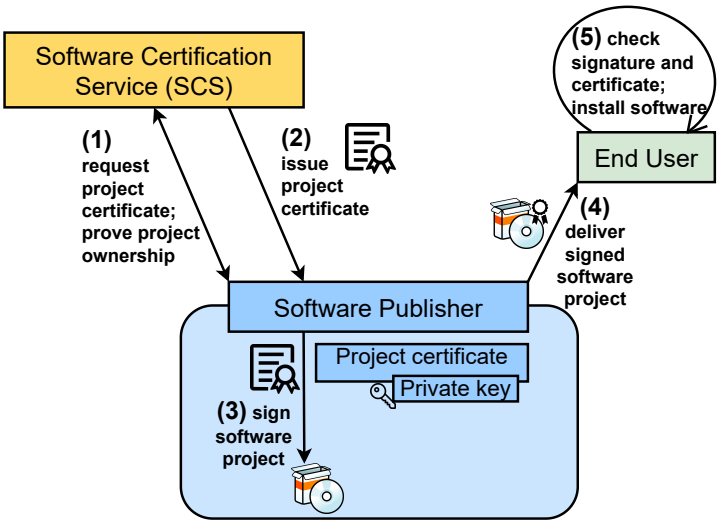


Figure 3.3 The software certification architecture.

when users download Firefox, they can verify that their copy of Firefox is the same one that is given to everyone else.

The main drawback of this solution is that its security guarantees have not been sufficiently scrutinized. For example, end users do not have a secure way to identify the domain used to certify the software and to establish trust in this domain. Further, the original implementation for Firefox has been phased out, along with their update to certificate transparency version 2 [80, 81]. It is not clear when – or whether – this binary transparency implementation will be used again. In fact, the last time a domain for fx-trans.net has been logged in a CT log has been in 2017 [82].

3.4 System and Threat Model

3.4.1 System Model

Figure 3.3 describes the general architecture of the proposed software certification service. At a high level, our approach is similar to the model employed by code signing. A *software publisher* contacts the *Software Certification Service (SCS)* requesting a

project certificate for a software project that it owns (*e.g.*, a Python package hosted on the PyPI community repository). The software publisher then proves ownership of the software project by executing a *software project certification* protocol with the SCS (Step 1). Once the certification protocol is completed successfully, the SCS issues a *project certificate* that binds together a *certificate public key* to a (project ID, project owner) tuple (Step 2). The software publisher then uses the corresponding private key to sign the software project (Step 3) for distribution to end users (Step 4). Finally, end users can verify the integrity of the retrieved software by checking the signature on the software and can get assurance that the software is authentic and originates with the project owner by checking the project certificate (Step 5).

The main difference from the code signing model is in Step 1. Whereas code signing seeks to establish trust in the identity of the software publisher based on a manual procedure that requires human intervention, our approach relies on a certification protocol that requires the software publisher to establish ownership of the software. The certification protocol is designed to be fully automated on the SCS side.

Software publishers that wish to apply for a project certificate need to establish an account with the SCS. This account will be used by the SCS to track interactions with the software publisher. During the software certification protocol, messages sent by a software publisher to the SCS server are authenticated using the publisher's *SCS account key*. Software publishers use a different set of credentials to manage projects hosted on a community repository, referred to as a *repository key* (*e.g.*, a password used to log into the community repository).

Community repository. We describe the salient features of a *community repository*, which hosts and distributes third party software that represents the main target for the proposed certification service. A community repository is a collection of individual projects which, usually, are open source and are developed using the

same programming language. For example, PyPI [56] (the Python package index), RubyGems [57] (the Ruby package manager), NPM [58] (the JavaScript package manager), or CPAN [83] (the Perl module manager).

Each project has a web-based homepage with a standard format that is uniform across all projects hosted on the same community repository. Typically, a project's homepage contains several sections that can be edited by project owner, such as the project name, owner details, project description, and download links. The proposed certification service leverages the *project description* section of a project's homepage during the protocol used to prove ownership over a project.

3.4.2 Threat Model and Security Goals

We assume that the SCS service will face adversaries that fit the following threat model. The SCS service itself (*i.e.*, the SCS server) is trustworthy and the private key used by the SCS service for signing project certificates is out of the attacker's reach. We also assume that a software publisher is able to protect her certificate signing key (this is the private key corresponding to the certificate public key). One way to achieve this is to store this key offline, and only use it to sign new project releases.

The communication between the SCS server and software publishers (acting as clients) happens over a secure channel (for example using SSL/TLS). We also assume that standard cryptographic primitives can be deployed, such as digital signatures that guarantee integrity and authenticity.

We consider the following types of adversaries:

A1: An attacker who gains access to the client's SCS account. This means that the attacker controls the SCS account key that is used to authenticate a publisher's messages to the SCS server. In this case, the attacker is able to impersonate a software publisher to the SCS service.

A2: An attacker who gains access to the project's repository account. This type of attacker controls the credential used by the project owner to manage the project on the community repository (*e.g.*, a password). This allows the attacker to arbitrarily change content in the project repository, including modifying the project description, adding/deleting project versions, or modifying the project files.

A3: An attacker who executes a network MITM attack between the SCS client and the SPCA server. This type of attacker may be a nation state that has the ability to tamper with messages exchanged between the publisher and the SCS service as part of the software certification protocol.

Although an A2-type adversary may gain access to a project's repository account, we assume that the attacker does not control the entire infrastructure of the community repository. As such, the attacker cannot cause the community repository to provide different views of the project repository to different sets of clients.

Attacker goals: The attacker seeks to obtain a valid signed project certificate that binds a tuple (Project owner ID, Project ID) to a public key PK, such that the attacker is not the owner of this project and it possesses the private key corresponding to PK. This will allow the attacker to sign arbitrary versions of the project (*e.g.*, a malicious version that has a backdoor embedded).

3.4.2.1 Security Goals.

Only the legitimate owner of a project should be able to complete a certification protocol for that project. Still, we need to account for occasional events when an attacker gains control over a project's repository, *i.e.*, we need to provide compromise resilience.

Of particular interest are adversaries that can gain control over a project for a short amount of time, during which they may attempt to obtain a project certificate by executing the certification protocol stealthily.

If, on the other hand, adversaries must maintain control over a project for a prolonged period of time in order to successfully complete the certification protocol,

this is arguable more difficult to achieve while going undetected. This is especially true if the certification protocol produces artifacts that are publicly visible on the project's webpage.

Concretely, we aim to achieve the following security goals in order to be able to withstand the aforementioned threat model:

SG1: Only an entity that controls an identifier should be able to successfully complete the certification for that identifier (by completing the given challenge). In particular, only the owner of a software project should be able to complete the certification protocol for that project.

SG2: Messages generated during one execution of the certification protocol for one account (*i.e.*, between the SCS server and one client) cannot be used towards obtaining authorizations for other accounts.

SG3: Attackers that gain control over a project's repository for a short period of time should not be able to successfully complete the certification protocol. This prevents an attacker that gains control over the project for a brief amount of time from obtaining a project certificate unbeknownst to the project owner.

SG4: Anyone who can access a project's webpage should be able to know whether an instance of the certification protocol is currently running for that project. In particular, the project owner should be able to tell if someone other than the project owner is trying to obtain a certificate for the project.

3.5 Protocol

3.5.1 Preliminaries

3.5.1.1 General Terms.

During the course of execution of the proposed protocol for software certification, we make use of the following terms:

- SCS server: The server software run by the Software Certification Service (SCS) acting as a Certification Authority (CA) that issues project certificates upon request by software publishers. We assume that the SCS server is available at `\scsurlbasic`.

- SCS client: The client software run by a software publisher that interacts with the SCS server in order to obtain a project certificate for a project owned by that publisher.
- Project Repository: The repository used for hosting the project for distribution. This refers to an individual project repository hosted on a community repository.
- Project: The project/package for which the certificate is requested.
- Project Owner: The software publisher who owns the project for which certification is requested. The project owner controls the SCS client and the project hosted on the repository.
- End-user: The users that download the project distribution from the project repository for installation and use.

3.5.1.2 Keys.

The SCS server has a *CA key pair pair*, and uses the CA private key to sign project certificates. The CA private key has high value and its compromise can have serious consequence for the security of the SCS service. As such, the CA private key must be kept offline, or protected using dedicated hardware (*e.g.*, HSMs).

The following types of keys are used by the project owner, acting as a client for the SCS server:

- *SCS account keys* (public/private key pair): Used to authenticate an SCS account holder (acting as a client) to the SCS server. Specifically, the client uses the SCS account private key to sign the messages sent to the SCS server while executing the SCS certification protocol. There is only one SCS account key pair per client, generated by the client. Once registered with the SCS server, an SCS account key can be used to obtain multiple project certificates for multiple projects owned by the client.
- *certificate keys* (public/private key pair): This key pair is generated by the client (acting as a project owner) and its public key is included in the project certificate generated by the SCS. The corresponding private key will be used by the project owner to sign a software project.
- *repository key*: This is the credential used by a project owner to manage the project on the community repository. For example, it can be the password used by the project owner to log into her account with community repositories such as PyPI, RubyGems or NPM.

3.5.1.3 High-level Details.

As our proposed protocol is inspired from the ACME protocol, we reuse several of ACME's protocol design choices. We mention these details here, so as not to overload unnecessarily the actual protocol description.

JSON objects and signatures. Information exchanged between the SCS server and clients is encapsulated in objects encoded as JSON messages [84] carried over HTTPS. Typically, the client sends to the SCS server a stub object, and the server returns the object where various fields have been filled.

Messages sent by the client to the server are signed using the private key of the client's SCS account key pair. The server uses the corresponding public key to verify the authenticity and integrity of messages from the client.

Nonces against replay attacks. To ensure protection against replay attacks, the protocol uses an anti-replay mechanism based on *nonces*: The server maintains a list of nonces issued to clients, and any signed request from the client must include a nonce. The server verifies that the nonces it receives from clients are among those that it has issued to clients, and ensures that nonces can be used at most once by clients.

Server-side resources. The SCS server offers services to clients via *server-side resources*. For example, a client can: send a request to the `newNonce` resource on the server in order to get a new nonce; send a request to the `newAccount` resource to create a new account; or send a request to the `newOrder` resource to initiate a project certificate order. The `directory` resource is used by clients to enumerate all available server-side resources. Server-side resources can be reached based on well-defined URLs. For example, the `newAccount` resource can be reached at `\scsurlbasic/scs/new-account` and the `newOrder` resource can be reached at `\scsurlbasic/scs/new-order`.

3.5.2 Certification Protocol Description

We now describe the protocol used by the SCS to issue a software project certificate. The protocol has two major phases:

1. Register an account with the SCS server
2. Request a project certificate

Phase 1 is carried out only once, when the publisher is communicating with the server for the first time. Each publisher creates an account with the SCS server, so that the SCS server can keep track of its interactions with various different publishers. The same account can then be used to get certificates for multiple projects owned by the publisher. Phase 2 is carried out every time the publisher needs a certificate for a project.

3.5.2.1 SCS Account Registration.

The protocol execution is initiated by the publisher (*i.e.*, project owner) using the SCS client. To register an account with the SCS server, a publisher emgages in the following protocol with the SCS server:

1. The client generates a fresh pair of SCS account keys (public/private key pair).
2. The client sends to the SCS server a registration request that contains the following information:
   - the contact details of the client (email address).
   - the SCS account public key.
   - a signature over the entire registration request using the SCS account private key.
3. The SCS server verifies that the signature is valid and that no account is already registered under this SCS account public key. The server then creates an account and stores the SCS account public key used to verify the registration request. This SCS account key is used to uniquely identify the account and will be used to authenticate future requests associated with this account.
4. The SCS server informs the client that the account was successfully created.

3.5.2.2 Obtaining a Project Certificate.

To obtain a project certificate, a publisher who has previously registered an SCS account, takes the following major steps:

1. Submit a project certificate order
2. Obtain authorization over the project identifier (*i.e.*, prove control over the software project for which the certificate is requested)
3. Finalize the order by submitting a CSR
4. Download the issued project certificate

We describe next the protocol interactions for these steps.

(1) Submit a project certificate order. The client sends to the SCS server a project certificate order request that contains the following information:

- software project identifier for which the certificate is requested (*e.g.*, project URL)
- certificate expiration date

Upon receipt of the order request, the SCS server performs some basic checks regarding the project identifier, such as checking the validity of the project URL. The server may also check if the project URL matches one of the participating community repositories.

The SCS server then informs the client that the order is created, together with an "`expires`" time by when the client needs to complete authorization of the requested project identifier. The server's response also contains an Authorization URL and a Finalize URL. The Authorization URL is a location on the server where the server makes available an identifier authorization resource associated with this new order request. The Finalize URL is a location on the server where the client will inform the server that it has completed the project ownership proof requirement.

(2) Obtain authorization over the project identifier . The project identifier authorization process establishes that an SCS account holder is authorized to manage project certificates for a given project identifier. For this, the client must prove ownership over the project by completing multiple validation challenges chosen by the SCS server. At a high level, to complete a validation challenge for a project, the client provisions the challenge response on that project's repository (more details in Section 3.5.2.3).

The following steps are executed in order to complete a validation challenge:

1. The client sends a request to the Authorization URL and the SCS server responds with an Authorization object that contains the following information:
   - the project identifier (*i.e.*, the project URL)
   - the Challenge URL
   - a validation token for this challlenge

   The validation token is a string randomly generated by the SCS server for this challenge. The Challenge URL is a location on the SCS server where the client will notify the server that the challenge has been completed.
2. The client completes the challenge by provisioning the challenge response on the project's repository.
3. The client notifies the SCS server that the challenge was completed by sending a request to the Challenge URL.
4. The SCS server verifies that the challenge was completed.

To address the threat model described in Section 3.4.2, the SCS certification protocol requires a client to respond to multiple challenges spread over time, and the SCS server to check that the client's response to the challenges remains persistently visible on the project's repository. In Section 3.5.2.3, we give more details on how challenges are completed by the client and verified by the SCS server.

(3) Finalize the order by submitting a CSR. Once the client believes it has completed the server's requirements for this project certificate order, it generates a certificate key pair (public/private keys). It also creates a Certificate Signing Request (CSR) which contains the client's requests with regard to the contents of the project certificate to

be issued. The client then requests to finalize the order by sending the CSR to the Finalize URL. The CSR contains the following information:

- software project identifier for which the certificate is requested (*e.g.*, project URL)
- certificate public key
- project owner details (name, email address)
- owner email address
- temporal information (valid from date, expiration date)
- a signature over the entire CSR using the certificate private key

If the request to finalize the order is successful, the SCS server issues the project certificate, which is signed with the server's CA private key. The SCS server then responds to the client with a Certificate URL.

(4) Download the issued project certificate. The client downloads the issued project certificate by sending a request to the Certificate URL, located on the SCS server.

3.5.2.3 Identifier Authorization.

An attacker who gains control over the project's repository for a brief period of time may be able to provision the challenge response on the project's repository, notify the server to validate the challenge, and then quickly remove the challenge response from the project's repository. In order to achieve security goal SG3 and mitigate attackers that can take control of the project repository for a brief period of time, we design the identifier authorization step to last over an extended period of time. In this way, a successful attacker would have to maintain control over the project repository for a longer period of time, which is arguably more difficult to achieve while going undetected.

Specifically, to obtain authorization over a project identifier, a project owner acting as a client in the certification protocol must complete not just one challenge,

but multiple validation challenges spread over an *identifier validation window* of time. Additionally, for each challenge, the client must not only provision the challenge response on the project's repository, but must also maintain persistently this challenge response on the project's repository over a *challenge validation window* of time. For example, we may consider a 7-day identifier validation window [1], during which the server will send a new challenge every 24 hours for 7 days in a row. For each challenge, the server will check the persistence of the challenge answer on the project's repository multiple times randomly during the 24-hour challenge validation window.

The project identifier authorization process establishes that an SCS account holder is authorized to manage project certificates for a given project identifier.

Validation of individual challenges. For each validation challenge, the client must provision a challenge response on the project's repository. In order to achieve security goal SG4 and deal with long-term adversarial presence, the validation requires that the response to a challenge must be placed on the project's repository in a publicly visible way. This will prevent an adversary to execute the certification protocol stealthily, as the legitimate project owner and/or other project maintainers will notice that a certification protocol is ongoing.

Specifically, to complete a challenge, the client must provision the challenge response in the *project description* section of the project's homepage. To preserve the functionality of the project description section and reduce confusion for the casual user who browses that project's homepage, the challenge response is placed at the end of the project description, using delimiters that make it clear they are not part of the actual project description. Placing the challenge response in the project description meets our requirement that the certification protocol must generate artifacts that are publicly visible.

[1]We picked 7 days based on previous repository breaches. The breaches were detected as early as a few hours in some cases whereas it took almost 5-7 days in other cases [85], [86], [87].

The client generates the challenge response as a Base64-encoded string of characters generated by concatenating the validation token for the challenge with a key fingerprint, separated by a "." character:

Response = token —— "." —— base64(fingerprint(SCS account key)),

where "——" denotes concatenation of strings, and the fingerprint is computed as a SHA-256 digest of the SCS account key. The response is placed at the end of the project's description, using clear delimiters:

——Response—— +=+ LoqXcYV8q5ONbJQx.bmR7SCTNo3tiA +=+

After notifying the server about completion of the challenge, the client needs to maintain the challenge response on the project homepage during the challenge validation window. The server calculates the value of the challenge response and checks the existence of this value on the project homepage multiple times at random times within the challenge validation window. If all the server checks during the challenge validation window are successful, the server deems the challenge as successfully completed, and generates the next challenge for the client.

3.6 Security Analysis

We now turn to analyzing the security of the proposed SCS protocol. We first show that the protocol meets the security goals stated in Section 3.4.2, and then analyze the protocol's compromise resiliency.

SG1: We need to show that only a project's owner should be able to complete the certification for that project. To prove ownership over a project, which is required in order to complete the certification protocol, an entity must successfully complete the challenges generated by the SCS server. As such, for each challenge, the entity must both:

- Hold the private key of the SCS account key pair used to respond to the challenge. This is because the responses from the client to the SCS server must be signed with that key.
- Control the project in question. This is because successfully provisioning the challenge response on the project's homepage requires write-access to the project's repository.

Since only the project owner has write-access to the project's repository, a successful execution of the SCS protocol provides the assurance that a specific SCS account holder is also the entity that controls a project (*i.e.*, the project owner).

SG2: We need to show that messages generated during one execution of the certification protocol for one account (*i.e.*, between the SCS server and one client) cannot be used towards obtaining authorizations for other accounts. This is achieved because all messages sent by an SCS client to the SCS server are signed using that client's SCS account private key. Thus, such messages cannot be reused between instances of the of the certification protocol executed by different SCS account holders.

The ACME protocol specification mentions a potential attack where a MitM attacker on ACME HTTPS requests can switch out a legitimate domain holder's account key for one of his choosing (described in Section 10.2 of the ACME RFC [76]). A similar threat exists for the SCS protocol. Similarly with ACME, we prevent such an attack by provisioning in all challenges a binding between an SCS protocol execution and the private key of the SCS account involved in that protocol execution: The challenge response reflects the public key of the SCS account key pair and is verified by the SCS server when validating challenges.

SG3: We need to show that attackers that gain control over a project's repository for a short period of time are not be able to successfully complete the SCS certification protocol. The certification protocol is designed so that the identifier authorization step lasts over an extended period of time. An entity attempting to complete the certification protocol for a project must complete multiple challenges. For each

challenge, the challenge response must be maintained persistently on the project's homepage, because the SCS server will check multiple times randomly during the challenge validation window. If an attacker is able to briefly gain control over the project's repository, she maybe able to provision a valid challenge response for that challenge. However, such an attacker will not be able to successfully provision valid information for subsequent challenges.

SG4: We need to show that an attacker cannot complete an SCS certification for a project in a stealthy manner. The SCS protocol achieves this by requiring that all challenge responses must be placed on the project's repository in a publicly visible way (*i.e.*, on the project's homepage). This ensures that the legitimate project owner and/or other project maintainers will notice that a certification protocol is ongoing.

3.6.1 Compromise Resiliency

We now analyze the SCS protocol resiliency against key compromise. As described in Section 3.5.1.2, the SCS service makes use of different keys for different purposes. As such, the impact of key compromise depends on which of these keys have been compromised.

3.6.1.1 Compromise of the Repository Key.

If an attacker is able to get hold of the *repository key* for a project, this allows the attacker unfettered access to the project repository, including making changes to the project's homepage. The attacker can register an account with the SCS server and then request a project certificate under this SCS account. Having access to the repository key, the attacker will be able to provision challenge responses on the project homepage.

The SCS protocol has two safeguards in place to deal with a repository key compromise. First, the certification protocol is designed to to last over an extended

period of time. Thus, if the repository key compromise is detected early enough, the project owner can change the repository key, preventing the attacker from successfully completing the certification protocol. In this way, a successful attacker would have to maintain control over the project repository for a longer period of time, which is arguably more difficult to achieve while going undetected. Second, the SCS protocol requires that the response to a challenge must be placed on the project's repository in a publicly visible way (*i.e.*, the project's homepage). Since the adversary has no control over the project homepage, they cannot prevent the visible display of challenge token. This will prevent an adversary to execute the certification protocol stealthily, as the legitimate project owner and/or other project maintainers will notice that a certification protocol is ongoing and will take steps to terminate such an active threat.

3.6.1.2 Compromise of the SCS Account Keys.

The SCS account keys are used to manage an SCS account. All the messages sent by the client to the server in the SCS protocol are digitally signed using the private key of the SCS account key pair.

If the attacker is able to get hold of the SCS account keys for a specific SCS account (*e.g.*, the SCS account of the project owner), then the attacker can interact with the SCS server on behalf of the legitimate SCS account owner. However, because the attacker is not in control of the project (*i.e.*, does not have the repository key for the project), the attacker cannot successfully complete the certification protocol.

Still, the attacker can try to inject messages in an ongoing certification protocol, or can try to abuse the SCS account management functions such as rolling over SCS account keys or deactivating the SCS account. Whereas such actions can have a short-term effect similar to a denial of service attack, they will not go unnoticed by the project owner who can take steps to restore the the security of the SCS account credentials.

3.7 Deployment

We have instantiated the SCS protocol with several practical deployments. In this section, we first provide general implementation details about the SCS service. We then present SCS deployments to automate four popular community repositories, PyPI [56], RubyGems [57], NPM [58], and GitHub [14]. Finally, we present an SCS deployment to automate the delegation process in community repositories that rely on systems like TUF [3, 4, 5] to provide compromise resilience.

3.7.1 SCS Implementation Details

The SCS service has two components, the server and the client. We implemented the SCS server on top of Boulder [88], which is an open-source ACME-based CA built for Let's Encrypt and written in Go. We have adapted the code to process the Project ID (the project repository URL) instead of the domain names. For example, when the client requests a project certificate, the server verifies that the project URL comprises of a valid set of characters and that the URL belongs to one of the community repositories that the SCS service has been deployed to. We also implement the challenge-response protocol used for proving project ownership. The SCS server is a stateful implementation that keeps track of the challenge-response process and how far the client is in the proof of ownership process. The process on the server side is automated and doesn't require manual intervention.

For the implementation of the SCS client, we tweaked Lego [89], which is an ACME client implementation for Let's Encrypt, written in Go. The SCS client is responsible for initiating the certificate issuance process, by placing a request to the server. The client is also responsible for participating in the challenge-response protocol and for fulfilling the challenge issued by the server. The project owner gets the challenge response from the SCS client and provisions it on the project homepage.

This is the only step that requires manual intervention during the challenge-response SCS protocol execution.

3.7.2 Deployment to Community Repositories

We deployed the SCS service to several community repositories to automate the issuance of certificates for the projects hosted on these repositories. By design, the SCS service does not require any changes to these community repositories, which makes it deployable right away and serves as an incentive for adoption.

The SCS service can be deployed to community repositories where each individual project has a dedicated webpage containing a project description section. Most community repositories fit this scenario, with the project description being normally used to provide basic information about the project. Although every community repository may have a different web layout for the project description, all the projects that are hosted on the same community repository have the same layout for the project description.

During the SCS protocol execution, the project owner needs to provision on the project description webpage the responses to challenges issued by the SCS server. To preserve the functionality of the project description field and reduce confusion for the casual user who browses the project's webpage, the challenge responses are placed at the end of the project description, using delimiters that make it clear they are not part of the actual project description (as described in Section 3.5.2.3).

Each time the project description needs to be updated, the project owner needs to upload a new version of of the project. The reason is that most community repositories do not allow altering metadata for a project version (including project description) once that version has been uploaded. As a side effect, at the end of the ownership proving protocol (which involves responding to multiple challenges), the project will have several versions that differ only in the project description. In

order to minimize this artificial increase in the version numbers due to fulfilling SCS challenges, we provide some recommendations for project owners. After completing the SCS protocol and downloading the project certificate, the project owner can hide these artificial versions that were added as part of the SCS protocol execution. This is known as yanking (for PyPI and RubyGems) or unpublishing (NPM and GitHub). Yanking/unpublishing a version means that when the end user tries to install the package using installers, the installer avoids the yanked versions. Alternatively, the project owner can leverage the version naming scheme used by the specific community repository. For example, the project owner can designate the artificial versions as post-release versions. It should be noted that changes in naming or managing the versions doesn't affect protocol execution in anyway, the suggestions presented here are just for the convenience of the project owner.

3.7.2.1 SCS for PyPI.

PyPI [56] is used for hosting and distributing Python packages. We use the "Project description" page to display the SCS challenge response. For this, the project owner includes the challenge response in the project description section of the *setup.py* file, which generally contains the metadata for the Python package, and then builds the package and uploads it to PyPI. As a result, the challenge response will be publicly visible to end users on the project's webpage, as shown in Figure 3.4.

3.7.2.2 SCS for RubyGems.

RubyGems [57] is used for hosting and distributing Ruby projects, known as "gems". To display the SCS challenge response, we use a section on a project's webpage where the owner can provide a short description of the project, which can range from a single sentence to a few paragraphs. Also, the project page does not allow HTML or Markdown formatting and so, unlike in PyPI, project owners do not have any

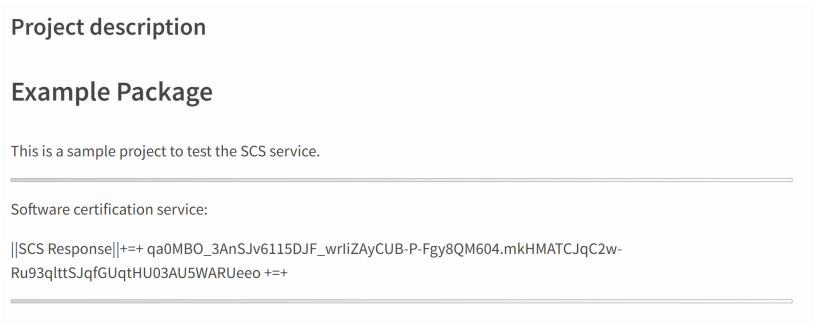


Figure 3.4 Challenge response displayed in the "Project description" section for a PyPI package.

choice in the way a challenge response gets displayed in the section. The project owner includes the challenge response in the description field of the *gemspec* file, which generally contains the metadata for the gem, and then builds the package and uploads it to RubyGems. As a result, the challenge response will be publicly visible to end users on the project's webpage, as shown in Figure 3.5.

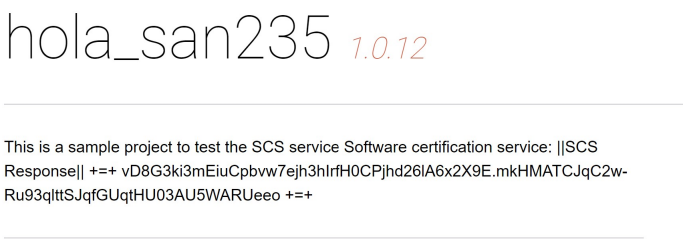


Figure 3.5 Challenge Response displayed on the project webpage for a RubyGems project.

3.7.2.3 SCS for NPM.

NPM [58] is the Node package manager used for hosting and distributing JavaScript packages. We use the "Readme" page to display the SCS challenge response. For

this, the project owner includes the challenge response in the *package.json* file and then builds the package and uploads it to NPM. As a result, the challenge response will be publicly visible to end users on the project's webpage, as shown in Figure 3.6.

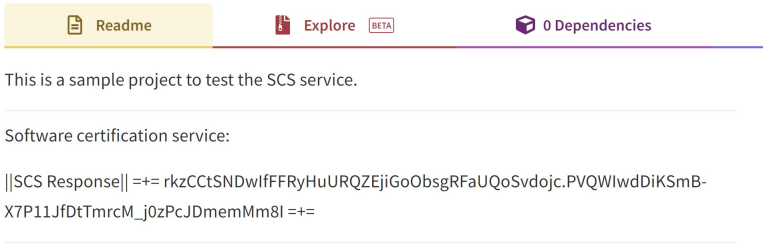


Figure 3.6 Challenge response displayed in the Readme file for an NPM package.

3.7.2.4 SCS for GitHub.

GitHub is used for version control and source code management. Unlike PyPI, NPM or RubyGems, GitHub is used to manage projects for multiple programming languages. For the projects hosted on GitHub, the part of the project that is most easily accessible and visible to end users is the README.md file for the project. We make use of this section for displaying the challenge response for GitHub projects. The SCS service currently works for all GitHub repositories that are hosted publicly and have a readme file, as shown in Figure 3.7.

3.7.3 Automating Delegations in Community Repositories

We consider a setting in which a system such as Diplomat [5] or TUF [3, 4] is used to provide compromise resilience for a community repository such as PyPI. This type of protection is achieved through several mechanisms, such as the use of *roles* (which allow to separate responsibility in a system) and *delegations* (which allow to distribute responsibilities in a system).

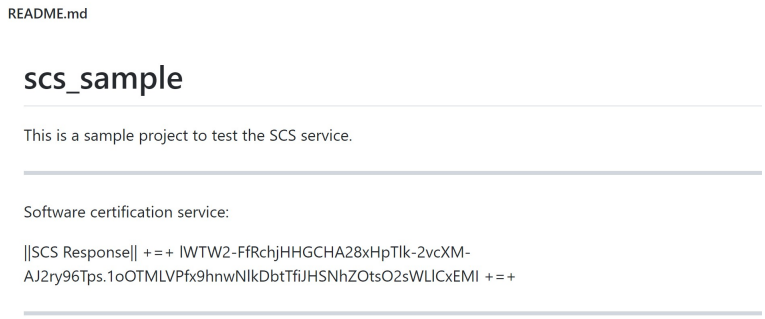


Figure 3.7 Challenge response displayed in the README.md file for a GitHub project.

For PyPI, there is a `root` role, which indicates which keys are authorized for other roles, such as the `projects`, `release`, and `timestamp` roles. The `projects` role is trusted to validate all the packages on PyPI. This role delegates trust for individual packages to the developers responsible for those packages. For example, the `projects` role may delegate the `BeautifulSoup` project to the public key belonging to the developer Alice, who is responsible for `BeautifulSoup`.

This delegation step can occur whenever a new project is created on PyPI, or an existing project wants to change an existing delegation. Currently, such a delegation involves manual operations on the part of the PyPI maintainers, which is not scalable considering that PyPI has over 345,000 projects (as of December 2021).

We automate this delegation step the SCS service. To have her public key certified as trusted for the `BeautifulSoup` project, the developer responsible for `BeautifulSoup` engages in the SCS ownership-proving protocol with the entity responsible for the `projects` role (*i.e.*, the PyPI server). If the developer successfully completes the SCS protocol, this serves as proof that the developer owns the `BeautifulSoup` project. As a result, the `projects` role will delegate trust for the `BeautifulSoup` project to the public key of this developer.

Specifically, once the ownership protocol is completed successfully, the server updates the top-level `projects` role to include a new 'delegations' entry to a new role `BeautifulSoupOwner` that is in charge of `BeautifulSoup`. This entry will include the public key of the developer responsible for `BeautifulSoup`. Then, the developer creates the `projects` file for the `BeautifulSoupOwner` role.

3.8 Related Work

3.8.1 Securing Community Repositories

Previous works studied the design and implementation of community repositories and proposed attacks [90], [91] and defenses [5], [92] [93]. These works focus on designing more secure software ecosystems with properties such as compromise-resilience and supply chain integrity. [94] discusses the security issues with the programming language specific community repositories like PyPI, RubyGems or NPM. In addition, due to the rising number of vulnerabilities and malware in the NPM ecosystem, various works [95], [96], [97] have been proposed to find new vulnerabilities, isolate untrusted packages, evaluate risks and remediate issues. [98] discusses the typosquatting and combosquatting attacks on the python software ecosystems like PyPI. These types of attacks have been on the rise in recent past. Most of the community repository specific work mentioned here, either deals with the vulnerability management in packages or with managing the dependencies in these packages. Our work differs from the previous work because our focus while designing the protocol is to provide guarantee about the project ownership to the end user. Thus ensuring that the end user can trust the project owner and verify the package before using it.

3.8.2 Software Certification

GPG signatures [99] have been used for trust management in community repositories for a long time. Most of the community repositories like PyPI, NPM, RubyGems etc. provide a feature that enables the developers to sign their packages before hosting them on the repository [100], [101], [102]. The GPG signature follows web of trust model. In this model, the developer generates the signing key and multiple users trust this key. The GPG signatures guarantee that the project has been signed by the developer and binds the developer to the project.

Code signing [75] and Binary Tranparency [79] are the mechanisms used for ensuring the users about the integrity of the software package. These mechanisms ensure the users that the package has been signed by a trusted developer and has not been altered after the release. These mechanisms bind the physical identity of the developers to the software project.

As discussed here, the earlier works in the direction of the software certification either ensure about the integrity of the software by answering the question *Is the signature still valid?* or give guarantees about the physical identity of the developers by answering the question *Is the developer trusted?* But, our work differs from these because we try to answer the question *Is the developer authorized to sign the project?* Thus, we are concerned with the binding between the Project owner, the project and the signature key. This contrasts with the existing mechanisms that are more concerned with the physical identity of the owner or with the integrity of the software.

3.9 Conclusion

In this chapter, we have presented a new approach for certifying the validity of software projects hosted on community repositories. Towards this goal, we have introduced a Software Certification Service (SCS) which gives software publishers the ability to prove the ownership of their projects and then get a project certificate that

binds the project owner and the project name to a public key. Although inspired from the ACME protocol in that it can be be fully automated on the SCS side, the proposed certification protocol is fundamentally different in its attack mitigation capabilities and in how ownership is proven.

We deployed the SCS service to several community repositories, including PyPI, RubyGems, and NPM, to automate the issuance of certificates for projects hosted on these repositories. By design, the SCS service does not require any changes to these community repositories, which makes it deployable right away and serves as an incentive for adoption. We are currently working with Google and PyPI on integrating our service into existing cloud security frameworks. As future work, we plan to extend the SCS service to more community repositories (currently, we require that each individual project has a dedicated webpage containing a project description section) and to explore other use cases that can benefit from automated verification.

CHAPTER 4

VERSION CONTROL SYSTEM FOR ARTIFACTS

4.1 Introduction

Adaptation of container technologies has changed how a software is built, packaged, and deployed. Containers are a packaging mechanism that provides the abstraction for applications from the environment in which they run. This abstraction allows applications to be deployed quickly and consistently, regardless of whether the target environment is a private data center, the public cloud, or even a developer's laptop. As a result of this flexibility and adaptability, there has been a massive shift towards adopting container technology for software deployment. Docker index statistics [103] show that activity in the Docker Hub registry has increased by 145% within a year. Due to the increased adoption of container technologies for software development, there has also been an increase in the development of tools and mechanisms that make container management, orchestration, and usage easier.

Deploying software using containerization generally involves using a container image. Along with a container image, a few other artifacts are also required to deploy software smoothly and efficiently. For example, a container image requires Helm charts [7] for smooth orchestration and open policy agent bundles [8] for access policies to secure the applications. There has been tremendous development in the storage and management tools for all these different artifacts required for software deployment. Each of these different artifacts serves different purposes, consists of different components, and use different storage registries and management tools. For example, the docker images are managed using registries like Docker Hub [104]. The helm charts and OPA bundles are managed using registries like Artifacts Hub [105]. Since different artifacts are managed differently, a project using multiple artifacts

needs to keep track of each separately and synchronize each when deploying the software. The process becomes cumbersome and confusing as and when the number of artifacts increases.

The Open Container Initiative (OCI) [106] solves this problem. OCI is an open governance structure for creating open industry standards for containers and related artifacts. OCI strives to create a uniform standard for managing container images. While registries like Docker Hub and Artifact Hub are used to maintain a particular artifact like Docker images or Helm charts for a project, other artifactories like JFrog [107] and Harbor [108] provide maintenance for all OCI artifacts in a single registry. These artifactories make the maintenance of different artifacts of a project more manageable. We also have tools and APIs like ORAS [109] that provide CLI features for pushing and pulling artifacts. These APIs make it easier for the end users to interact with the registries.

One area where these OCI compliant registries are lacking in terms of unified standards is version control. Currently, there is no existing uniform version control system for the artifacts. At present, versioning practices for these artifacts are decided by maintainers or developers. Every project has its way of denoting versions. Even the registries used for storing the artifacts have their methods for version control. For example, the Docker Hub provides tags that can be used for denoting versions of images. On the other hand, Artifact Hub does not use tags, but the versions are denoted for Helm charts manually as part of the metadata. In addition, some projects use source code repositories to keep track of configuration files separately. As a result of diverse practices for versioning, there is no uniform way to version control the artifacts. Consequently, as the number of artifacts increases for a project, it becomes increasingly difficult to keep track of changes. A uniform version control system for artifacts would make it easier for the users to keep track of changes to the artifacts and thus maintain a history of changes over time. Having a version control system in place

also allows users to roll back to earlier versions of the artifacts in case of unwanted changes or discovered vulnerabilities. Having a version control mechanism also helps with keeping track of the artifact as the number of versions of artifacts increases. As discussed earlier, existing mechanisms work differently for different artifacts and cannot be adopted as uniform practice across all artifacts. This necessitates a uniform standard for version control of OCI artifacts.

In this chapter, we propose a uniform version control system for artifacts. We examine the features of the artifact registries like Harbor, Artifact Hub, Docker Hub, and others to see if we can leverage any of the existing features for version control. We also examine the state of the art practices to find out why none of them can be adopted as a uniform standard for version control. Our primary focus here is artifacts recognized by OCI standards. We treat artifacts as structured objects with multiple components like file systems, binary packages, and metadata instead of treating them as just opaque binary objects. In order to do this, we analyze the structure of these artifacts to figure out the structure and components that make up these artifact binaries. We further leverage this structure and design a diff algorithm that computes the difference between two versions of artifacts. We also examine challenges related to the concurrency of operations, the persistence of older versions of artifacts, and the security of versions stored on the server. Finally, we design commit and update mechanisms that address these challenges. With the proposed commit and update protocols in place, the OCI artifacts can be version controlled uniformly. Lastly, we implement the proposed design for an open source, OCI compliant registry, Harbor, to demonstrate how the design works.

4.2 Background

4.2.1 Open Container Initiative (OCI) and OCI Artifacts

OCI provides uniform specifications for container images and container runtimes, enabling the developers to treat all artifacts uniformly irrespective of their structure and purpose. OCI provides a way to leverage OCI registries for arbitrary artifacts without having to treat them as container images. This project is known as the OCI artifacts project [110]. Artifacts project provides uniform specification and runtime for container images. In addition, using standards provided by OCI artifacts, other artifacts like Helm Charts for deployment and OPA bundles for policy management can also be stored in the same registries. Each of the artifacts supported by OCI is discussed in this section, along with their purpose and component structure.

4.2.1.1 Docker Images.

A Docker image is a read-only package containing instructions for creating a container that can run on the Docker platform [111]. These instructions provide a convenient way to package applications and server environments, which can be used privately or shared publicly with other Docker users. A Docker image consists of a collection of files that bundle together all the essentials, such as installations, application code, and dependencies required to configure a fully operational container environment. A docker image pulled from Docker Hub has applications, file systems, and dependencies that are part of the built docker image. In addition, it also has a plain text file called docker file. This file contains configurations required to run the docker image and build a running container from that docker image. In compliance with OCI standards, the image is accompanied by an OCI image manifest. Manifest is again a text file that contains metadata related to the image.

4.2.1.2 Singularity Images.

Singularity images are used for running complex applications on HPC clusters in a simple, portable, and reproducible way. A Singularity image, referred to as a "container" is a single file containing a virtual file system. After the image is created, operating system and applications can be installed and metadata can be saved. Unlike Docker which assembles images from layers stored on the local machine, a Singularity image is just one file that can sit on the local machine, or in a folder on the cluster, or anywhere. Having Singularity containers housed within a single image file greatly simplifies management tasks such as sharing, copying, and branching of containers.

4.2.1.3 Helm Charts.

A Helm chart [112] describes how to manage a specific application on Kubernetes [113]. It consists of metadata that describes the application and the infrastructure needed to operate it in terms of the standard Kubernetes primitives. Each chart references one or more container images that contain the application code to be run. Helm charts contain a package description (chart.yml) and one or more templates containing Kubernetes manifest files. Helm charts indicate to Kubernetes engine how to perform the application deployment and manage the container clusters.

4.2.1.4 OPA Bundles.

The Open Policy Agent (OPA) is an open-source, general-purpose policy engine that enables unified, context-aware policy enforcement across the entire stack. OPA provides a high-level declarative language to the author and enforces policies across the stack. In addition, OPA bundles the policy documents in bundles. Bundle files are gzipped tarballs that contain policies and data [114]. The data files in the bundle must be organized hierarchically into directories inside the tarball. The bundle contains a policy file and data files. The bundle may also contain an optional .wasm binary

file. It stores the WebAssembly compiled version of all the Rego policy files within the bundle. Bundle files may contain an optional .manifest file that stores bundle metadata.

4.2.2 How are Artifacts Related?

Each of the above mentioned artifacts is used at different stages of the container lifecycle [115]. First, the Docker or Singularity images are used to build containerized applications. Then, when the end-user deploys the images, they can run containerized applications on their system. Usually, docker-compose files are used for running these containers, but Helm charts make deployment easier for complex applications. As a result, the Helm charts are used to deploy these images using Kubernetes. Helm charts are equivalent to docker-compose in terms of usage but much easier to maintain and use. Thus, Helm Charts are used in the orchestration and deployment phase of the container life cycle. Finally, the OPA bundles enforce security policies like access control or authorization when the container runs.

Generally, each of these artifacts has public and private repositories used for managing them. Docker Hub [104] is commonly used for Docker Images, and Singularity Hub [116] is used for Singularity images. Artifact hub [105] is used for Helm charts. However, since OCI adopted the Artifacts project, there has been shift towards OCI compliant registries that can store and manage all these artifacts. As a result, all project artifacts are stored in an OCI compliant registry in a single project repository. Thus, a project repository in an OCI compliant registry has Docker image or Singularity image for the project, Helm chart used for deploying the image OPA bundles for the project that define the policies for the project. Harbor [108] is an OCI compliant registry that supports storing all project artifacts in the same project repository.

4.3 Benefits of VCS for Artifacts

In this section of the chapter, we enlist the benefits of having a uniform version control system for the OCI artifacts. These features are important to have in order to manage artifacts efficiently and deploy software smoothly.

Benefit 1. A VCS keeps track of the complete history of changes to all the files in a code repository. When it comes to artifacts, the registries either lack this feature or provide it partially. For example, Docker hub keeps track of history related to author, time of publishing the image and tags. But there is no history of what exactly changed between the two versions. It would be helpful for the end users if registries can also keep track of changes in terms of artifact file system, configuration files in addition to metadata.

Benefit 2. A VCS provides ability to rollback to earlier versions when required. Therefore, if the artifacts are version controlled, it is possible to rollback to earlier versions whenever the need arises. For example, if performing a component upgrade in a docker image breaks the whole environment, version control makes it easier to rollback to an earlier version of the image that is working. At present, Docker and other artifact registries do not provide the rollback/ revert functionality.

Benefit 3. The VCS for source code tells the user what exactly changed between versions. Currently, Docker hub and other artifact registries do not do this for OCI artifacts. The images and other artifacts are used as building blocks for other artifacts. Docker images and other artifacts might not be in human readable form, but still they are further used to generate other artifacts like running containers or Helm charts etc. For this reason, it is important that the end users know what exactly changed between two versions of artifacts and how it will reflect in the resulting artifact built using these images or charts. In addition to the ease of maintenance and use, a version control system can also be useful for detecting and mitigation of backdooring attacks

on OCI artifacts. When the VCS computes the diff and displays the changes in file system or configuration it is easier for the user to notice a change that is unexpected and thus be alert about the possible attacks.

Benefit 4. There is no uniform system for versioning of artifacts. Some of the registries use tags to denote versions, while others make use of configuration files to denote versions manually. This leads to confusions while deployment of artifacts because of the absence of a standard versioning system. An example of an incident caused by this confusion is highlighted in [117]. The confusion is caused because registries allow reusing the tags or manual numbering of versions becomes confusing as and when number of versions increase. Due to this reason, having a proper versioning system will help avoid errors caused by confusion between different artifacts using tags for the maintaining versioning information. It takes away the ambiguity around naming of versions, decisions around what part of versions do you store in the repository (as in just the changes or the entire repository). Similarly, when a VCS is used for managing artifacts, the ambiguity that currently exists around versioning schemes, tag usage etc. is removed and management of project artifacts becomes easier.

Benefit 5. Having the OCI artifacts under a proper VCS will allow the owners to keep the artifact repositories in sync and not allow the collaborators to overwrite the changes to previous versions when they try to push a new version to the repository.

4.4 Challenges in Designing a VCS for Artifacts

In order to design a uniform version control standard for artifacts, there are some design challenges that we need to address. In this section, we describe these challenges and why it is so important to design a VCS that addresses these challenges.

Concurrency. In a VCS, multiple users are using the same repository simultaneously. In order to handle multiple clients pushing at the same time and to

maintain atomicity and consistency of operations, Harbor makes use of a concept called optimistic concurrency control [118]. Optimistic concurrency control (OCC) is a method applied to transactional systems such as relational databases software transactional memory. OCC assumes that multiple operations can frequently complete without interfering with each other. While running, these transactions use data resources without acquiring locks on shared resources. Before writing data to the database, each write operation verifies that no other operation has modified the data that was read. If the check reveals conflicting modifications, the writing operation rolls back and can be restarted. Harbor makes use of this concept in order to ensure push and pull operations to the registry are in sync with each other. OCC enabled database maintained by harbor for storing artifacts means that the data written to the operations are not over written by other operations and the database remains consistent. We also intend to leverage this concurrency handling mechanism for the commit operations in the proposed VCS.

Garbage collection. For a VCS providing facility to rollback to older versions, it becomes necessary to keep all the older data in the database. This allows the server to retrieve any older versions when requested for. At present, Harbor has a garbage collection mechanism [119]. Garbage collection can either be scheduled or can be initiated manually for a repository. When initiated, the garbage collection deletes all data from the database that has been marked for deletion. This would mean that even if particular data is required to be preserved for older versions, if it is marked for deletion, it is removed during garbage collection. In order to provide features like rolling back to earlier versions, this needs to be changed. When garbage collection is initiated, the system should only delete data that is not referenced by any other artifact manifest. This means only database entries that do not have any references from other artifact manifests or are orphaned are marked for deletiona nd others are

not. It is important to implement this change so that older versions of artifacts can be preserved and retrieved for future use.

Binary Artifacts. One of the main reasons that OCI artifacts lack a uniform standard for version control is that the artifacts are treated as binary objects. This means that they are treated as opaque byte strings having no structure. As a result, the traditional VCS are not able to manage artifacts. In order to efficiently manage artifacts, it is necessary to treat these artifacts as structured objects that comprise of components and follow a particular structure. The main motive behind this consideration is the fact that, even though artifacts are built from source code, they are more often than not used as building blocks for other artifacts too. For example, Docker images are used to build running containers. For this reason, it is important to understand their structural components and build a VCS that treats them like building blocks and not end products.

Securing the Versions. When storing multiple versions of artifacts, it is important to consider the threats to security of the data that is stored by the system. At the same time, it is important to ensure that the end user is guaranteed that the data that they are retrieving from the system (*i.e.*, older versions) is exactly what they asked for and is not tampered with in any manner. In order to do this, we need to introduce signature mechanism to ensure integrity and authenticity of the data at rest in the version control system.

4.5 Current Practices for Versioning in OCI Artifacts

In this section, we will look at different practices adopted for versioning different artifacts and also see how these practices are not adequate for resource versioning of artifacts.

4.5.1 Versioning Using Existing Version Control Systems

An undeniable question arises, why can we not use the popular version control systems like Git and SVN for managing versions for Artifacts. To answer the question, it is essential to understand the difference between artifacts and source code. VCS like Git [10] and SVN [11] are designed to manage source code version control, usually a collection of human-readable text files. As discussed in Section 4.2 that OCI artifacts are Binary files that are further used to build other artifacts like running containers. Since traditional VCS-es like Git cannot handle binary files, we must look into the VCS-es used to maintain the binary files. Some of the popular VCS-es used for managing binary files are:

- JFrog Artifactory [107]: This management system is used for different types of OCI artifacts. It provides all management features but does not support version control like Git. The only way versioning information can be attached is by using the version field in metadata.
- CloudRepo [120]: This is also an artifactory, providing features similar to JFrog. But it is cheaper to use in terms of prices. However, it stores projects developed in Maven and Python only.
- Git LFS [121]: This is an extension of Git that can be used to store large binary files. Git Large File Storage (LFS) replaces large binary files such as audio samples, videos, datasets, and graphics with text pointers inside traditional Git while storing the file contents on a remote server. The difference between Git and Git LFS is how the diff is computed for the files. Git uses diff algorithms applicable to text while Git LFS uses ones applicable to binary files.
- Boar [122]: Boar stores snapshots of directory trees in a local or remote repository and provides tools to ensure that your data is consistent and complete. The user can keep just some or all of the data checked out for viewing and editing. In addition, the repository has a simple layout to ensure that the data can easily be extracted even if the original software should be unavailable. This simplicity makes Boar ideal for data that needs safe long-term storage.

The general convention about the version control systems is that the version control systems are used for keeping track of any text based files that are human-readable. The files produced from source files are not generally managed through version

control. As a result, the general convention is to manage configuration files used for generating images and other artifacts using Git like VCS. Then the actual artifacts are not part of the version control. This convention does not work because the images and other artifacts are used as building blocks for other artifacts. Docker images and other artifacts might not be in human-readable form, but they are still used to generate other artifacts like running containers or Helm charts. For this reason, the images and chart templates need to be version controlled. The version-controlled images make it easier to deploy the correct version of containers. As we see in all the VCS-es discussed above, the objects are treated as binary files. As a result, while storing it on the server or using it later, the user has no idea what changed between the two versions of the same artifact. The only information the user has is that some things changed, and hence there is a new version. Most maintainers would manually keep track of changes using changelog or similar techniques. However, this is not always ideal when the artifacts change frequently.

4.5.2 Versioning Using Configuration Files

As discussed in Section 4.2, all the OCI artifacts are packages consisting of binary files, configuration files, and metadata files. Configuration and metadata files for an artifact are generally text files (dockerfile for images, charts.yaml for helm charts, recipe file for singularity images). Thus, a common practice among developers is maintaining configuration files as part of source code repositories using version control systems like Git. Each artifact uses this concept differently.

Docker Images. Docker images are built artifacts stored in Docker Hub or similar registries. The Docker images are built from Docker files. These Docker files contain the script of instructions that define how to build an image. The built docker image is then published on Docker Hub. Every time there is a change in the docker file, the image needs to be built again and published. This means that the changes in

the docker file change the version of images pushed to the docker hub. As a result, a prevalent practice is to version docker files for the docker image and decide the version of the image based on that. To version docker images, either docker file is added to the source code repository of the application or the developers have a separate source code repository for just the docker files of the application, and they get versioned separately.

But none of these methods are adequate for properly versioning the docker images. Maintaining docker files in the same repository as the source code does not always reflect the most updated version of the docker file. The reason is that the docker file may change even if the source code has not changed. There may be an update to the base image or some other dependency. In this case, the docker file may change in this case, but the source code does not. In cases like this, the practice of maintaining a docker file, along with source code does not work properly.

Helm Charts. The helm chart version is indicated by the version number included in the charts.yaml file. This file has two types of version fields. One is the application version, and the other is the chart version. The two fields are not related to each other. *i.e.*, a change in one doesn't guarantee a change in the other. The application version is the version of the application that the chart deploys. The version of the chart is the version that the chart currently is in. Any change in the deployment configuration will change the version of the chart. The developers generally follow versioning practices that manage application versions and chart versions separately.

Singularity Images. The practices for versioning singularity images are very similar to that for Docker images. The file with all build configurations in singularity is called a "recipe file." The recipe file might be versioned separately and maintained like a docker file.

4.5.3 Versioning Using Tags

Since there is no well defined method of using tags to denote versions of artifacts, the users have their own conventions and standards to denote versions using tags. This leads to multiple ways of using tags in Docker and other registries to denote versions. Tags are most commonly used for versioning in docker images. Some of the widely used tagging schemes [9] in Docker images are described below.

Stable Tags. Stable tags mean that the user can keep pulling using the same tag from the registry, the image pointed to by the tag changes and keeps getting updated. The user doesn't have to worry about that. For example, "latest" tag might be used for denoting the latest the user keeps pulling the latest tag and gets the latest image, which changes over time, but the user doesn't have to worry about that. In this scenario, the developers decide when they want to update the image associated with the tag. Since the same tag points to different images over time, the tags are reused in this scenario.

Unique Tags. In this scenario, every image that is pushed to the repository, has a new and unique tag, which is never reused.Thus, the user has to pull the image using a different tag every time. The developers have to be careful about not reusing the tags.

There are a few alternatives that are considered for tagging images using unique tags:

- Git Commit: This option uses the git commit id as tag for every newly pushed image. This option works well until there is no update to the base image. The reason being that, when there is an update to the base image, the git commit id does not change, but the content of the image being pushed to docker hub changes. This leads to confusion in the versioning of images.
- Date-time-stamp: This approach uses timestamp as a tag. This is a common approach as this can inform the user when the image was built.

- Digest: the digest for an image is unique but it is very long. Thus, this option gives unique tags but makes them less usable.

One very important thing to be considered here is that tags are not immutable and can be changed or deleted anytime by the repository owner. If the tagging schemes are not well defined and practiced properly, it may lead to confusion in deployment of the application. It may lead to applications crashing during production [123]. There have been incidents of images being broken and services being unavailable to users because of confusions in the use of tags. In March 2018, node.js docker images were broken because of a problem with yarn [124]. Since the version of node.js had not changed, the tags did not change. As a result, the images with the latest tag that were working earlier stopped working when pulled again. Turns out that the problem was caused due to use of stable tags for denoting versions [117]. In another incident, in June 2018, LightTag which is a text annotation tool faced issues about availability due to stable tags used for versioning of images [125].

4.6 Internal Structure of the Harbor Registry

The VCS proposed in the chapter is designed with Harbor registry in mind because Harbor is an open source OCI compliant registry. In order to design a VCS based on Harbor registry, it is important to understand the internal structure of Harbor. At present, Harbor provides features to store different OCI compliant artifacts. Harbor is a distributed and open source artifact registry. At present, Harbor allows pulling, pushing and storing of artifacts. These artifacts are managed using individual *artifact repository*. All of the repositories for artifacts belonging to same project are part of same *project repository*.

4.6.1 Storage Structure of Harbor

Harbor registry's internal storage structure has three different components that handle artifact storage and management.

- k-v storage: This layer provides data cache functions and supports temporarily persisting job metadata for the job service.
- Data storage: This is the layer responsible for data persistence as backend storage of the registry. Data storage is the layer that stores the blobs, files, and objects that are associated with artifacts or projects.
- Database: This layer stores the related metadata of Harbor models, like projects, users, roles, replication policies, tag retention policies, scanners, charts, and images. PostgreSQL [126] is the database used by Harbor.

4.6.2 Data Structures in Harbor

Blobs are the primary storage unit for any artifact for a project. The data storage layer in Harbor stores the artifacts as blobs (binary objects). Harbor uses two types of blobs to represent data for a project in the database [127]

Artifact Blobs. The artifact blob represents an artifact in the Harbor Data storage. For every artifact associated with the project, a blob is present. A new artifact blob is computed whenever a new artifact is pushed to the repository. A unique digest of the blob identifies the artifact blob. An artifact blob is a structure that consists of the Artifact, Name (string), Snapshot (timestamp when the blob was created), Blob Properties (type, creation date, last modified, Tag) The blob digest for an artifact is computed by hashing over all the structure details. After computation, the blob is stored in the database along with the Blob digest, Content type (*i.e.*, artifact type), Size, Update time (*i.e.*, last update time)and Creation time. Figure 4.1 shows how Artifact blob is represented in the database and how the digest is computed.

Project Blobs. Project blob represents the project repository in Harbor. A project blob represents a project's state at any particular time. It is a snapshot of the project. A project blob is a structure with the Artifact blob digests for a project, Manifest for the artifacts, Name, Creation time, Snapshot (timestamp when the blob was created), Project properties (owner (email ID), creation date, last modified). The artifact blob digest and manifest for all artifacts is a list ordered by the timestamp associated with

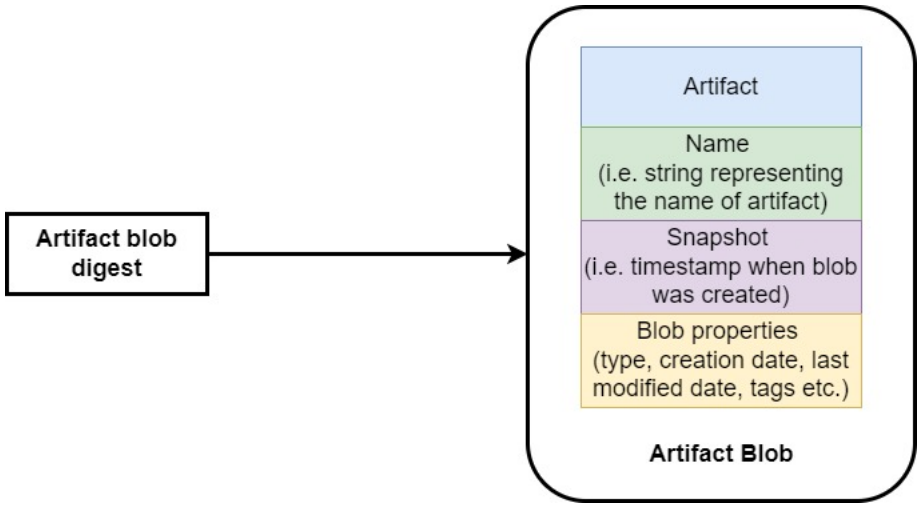


Figure 4.1 Structure of Artifact Blob in Harbor.

the artifact blob. A project blob digest is computed over the above mentioned details. This digest is a uniquely identifying key for the project. After creation, the blob is stored in the database along with the Digest, Creation time, Update time.

4.6.3 Creation and Deletion of Blobs

When the client initiates a push operation for pushing an artifact, a new artifact blob is created corresponding to the new artifact. Since a new artifact blob is created, a new project blob is created, reflecting the change in artifacts related to the project. When deleting an artifact, the blob corresponding to that artifact is marked as deleted in the database but is not actually deleted from the database. Also, a new project blob is computed to reflect changes in the artifact associated with the project. Harbor has a garbage collection mechanism that the client can invoke to delete the blobs that are no longer referenced by any other blobs. When garbage collection is initiated, the blobs marked as deleted are deleted from the database. The garbage collector can also be configured only to delete blobs with timestamps older than a particular value.

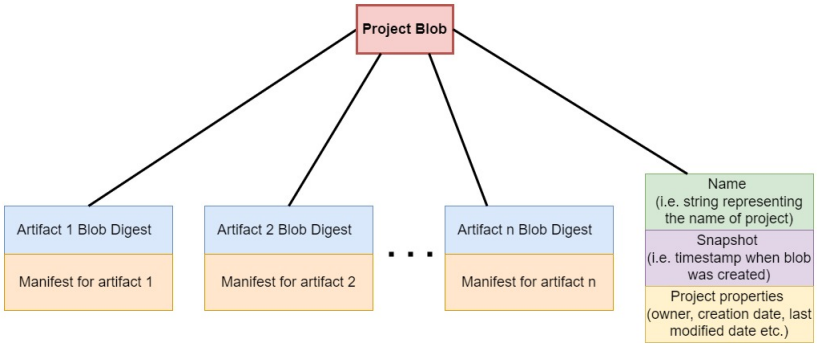


Figure 4.2 Structure and computation of Project Blob in Harbor.

On project deletion, all the blobs associated with the project and artifacts for the project are marked for deletion.

4.6.4 Security Features in Harbor

When storing deployment artifacts in the registry, it is important to ensure the integrity and authenticity of the data stored. At present Harbor has the following mechanisms to enhance the security of the artifacts:

- **Role based access control in repositories**: This feature allows the repository owners to decide and delegate roles to other users of the repository. Access control decides what features can be used by users and what functions can be performed.
- **Content Trust for docker images**: This feature allows the users of docker images to sign docker images ensuring integrity of images managed using Harbor Repository.

4.7 Internal Structure of Proposed VCS

4.7.1 Storage Structure

In order to accommodate version control features, we make changes to the internal storage structure described in Section 4.6.1. We change the data storage layer that

stores blobs for artifacts. We also make changes to the database layer in order to manage multiple versions of the artifacts and their storage.

In order to store multiple versions of artifacts in the repository and maintain a relation between these versions, we introduce a new data structure in addition to artifact blob and project blob. We introduce *Commit blob* which keeps track of versions committed for every artifact. There is a commit blob for every version of an artifact.

Commit Blobs. A commit blob and keeps track of details about what data was committed, when it was committed and who committed it to the server. A commit blob is created whenever a new artifact is pushed to the repository and a project undergoes a change. A commit blob is a structure consisting of the Project blob digest, Artifact blob digests that changed, Commit blob digest for previous commit, Creation time, Author. A commit blob digest is computed over these details. The commit blob is stored by the server along with the Commit blob digest, Creation time and Author. The blobs are identified uniquely by their digests and they are immutable. This means that once the blob is created, it cannot be changed. Every new push/ commit to the repository results in creation of new blobs. Along with the blob, signature over the blob is also computed. This signature is also stored in the database. The signature ensures that the data in the commit blob cannot be tampered with. This signature is computed every time a version is committed. It is verified every time a version is requested from the server. Figure 4.3 shows the structure of proposed commit blob and computation of signature over the blob.

Thus, the VCS architecture now has three types of blobs in the local storage. The commit blob is computed every time a new version of an artifact is committed. A commit object encompasses the data for project blob and artifact blobs associated with that commit. A commit blob digest identifies each of the version committed.

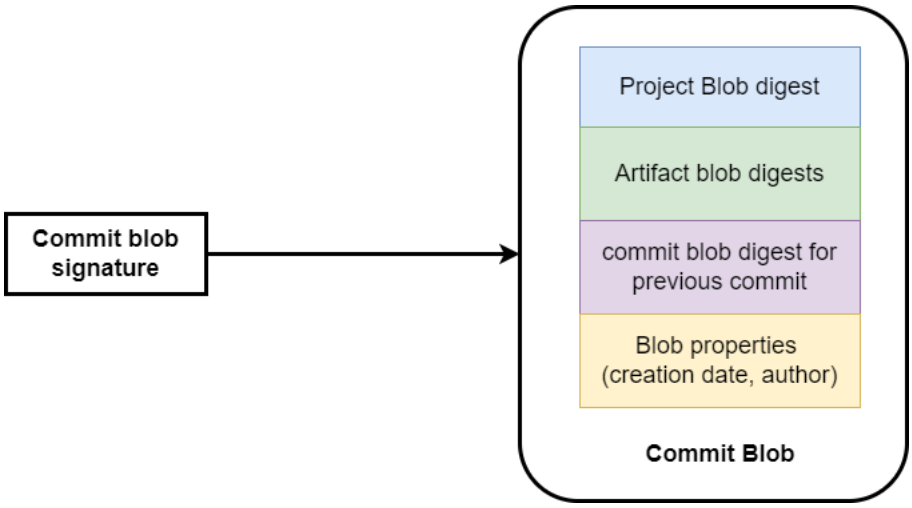


Figure 4.3 Structure of Commit Blob and computation of signature.

4.7.2 Security Features in VCS

Since we are storing new type of data structure and storing more data on the server, it is necessary to ensure the integrity and authenticity of this data. In order to ensure integrity and authenticity of data that is committed to the server, we introduce signatures over commit blob. This signature is computed over the commit blob computed by the client for every commit. This signature with the virtue of being computed on the commit blob, the signature encompasses details about project blob and artifact blobs related to that particular commit. The signature also secures details about the previous commit. Thus, when the signature is verified, the user can be assured that the data associated with the commit has not been tampered with. Also, the data related with previous commit has not been tampered with. Thus, the user can be assured that the data committed to the server is not tampered with. The commit signatures also assure that the data received by client during update operations is exactly what the client had asked for.

4.7.3 Commit and Update Protocols

Once the changes have been made to local copis of artifacts, it is necessary to publish these changes so that other maintainers using the same repository are aware of the changes. At the same time, it is necessary for the end users to update their local copies to the latest versions to reflect changes published by other users. For this purpose, we propose commit and update protocols. This section describes the proposed commit and update protocols.

4.7.3.1 Commit Protocol.

The commit protocol is initiated by the client when the client needs to publish a new version of the artifact on the server. The protocol can be described using the following steps:

1. The client(C) computes the diff for all the artifacts that have changed and need to commit changes to the server(S).
2. C computes the artifact blob for each of the artifacts that changed.
3. C computes the project blob for the project.
4. C computes the commit blob for the project.
5. C also computes signature over the commit blob computed in Step 4.
6. C then sends the artifact blobs, project blob and the commit blob and the signature to the server.
7. The server stores these blobs for future use.

4.7.3.2 Update Protocol.

The update protocol is initiated by the client when the client needs to get the latest version of the artifact in the local repository.

1. C places a request with the server for the latest version of the project. For this, the C sends to S the commit id for the revision locally present with the client.

2. S gets the artifact blobs, project blob and the commit blob and signature over commit blob for the requested version.
3. S sends to C the blobs for the latest version to C.
4. C verifies the signature over the blob.
5. On successful verification of signature, C replaces the local version with the new one and informs the user about what exactly changed in the new version with help of the diff.

4.7.4 Artifacts as Structured Objects

In this chapter, we propose an approach to treat OCI artifacts as structured objects and not just opaque binary objects as version control systems currently do. The primary reason behind this is that the OCI artifacts are further used to build other artifacts like running containers and Kubernetes clusters. In this case, the end users of the artifacts must know what forms the artifacts and what changes between two versions of the artifact, so they can also know what changes in the resultant built artifacts. More importantly, the structure we define must be independent of differences in their package structures. This enables the VCS to be adaptable for any new artifacts adopted by the OCI artifacts project.

In order to define such a structure, we first need to study the structure of various OCI artifacts. Looking at the various OCI artifact structures, following components stood out as common among them:

4.7.4.1 Packages/Dependencies/File System.

Different artifacts require resolving different dependencies from the third party to build the resultant product. In the case of container images like docker images or singularity images, these dependencies might be third-party container images used for different layers of the current image. The dependencies might also be in the form of third-party packages like pip, npm, and apt packages used by the images to build a running container. For artifacts like helm charts, the dependencies are other charts

that the current deployment chart has to use to deploy the container successfully. These dependencies are generally denoted by the path describing their source. The build script resolves this path and makes use of the dependency accordingly. When given details about what dependencies the artifacts use, the end-users can make an informed decision about updating to a newer version or rolling back to earlier ones. On the other hand, suppose the end-user is unaware of the artifact's dependencies or the changes in dependencies for newer versions. In that case, the attackers can easily replace the dependencies and execute dependency confusion attacks. The dependencies listed in artifacts are binary files/packages.

4.7.4.2 Configuration Files.

Each OCI artifact has a configuration file, which is generally a text document. The configuration file contains the commands for building the artifacts. For example, the container images have dockerfile or recipe files, helm charts have YAML files containing configuration details, and the OPA bundles have similar files. Since these files contain build instructions, the changes in these files would mean that the resultant build will also change. , In this case, when in an updated version of the artifact, the configuration files change, and the end users should be aware of what configurations have been changed.

4.7.4.3 Metadata.

Each OCI artifact, in addition to having configuration files, packages, and dependencies, also has metadata related to the artifacts. The metadata includes details like artifact size, artifact publishers, and maintainers. The artifact manifest file contains these details.

4.7.5 Diff Algorithm

The Diff algorithm that we use for the OCI artifacts should be able to compute the diff between two versions of the OCI artifacts not in the form of byte strings but in terms of differences in file systems, configurations and metadata. Thus, we need to design an algorithm that works for all these scenarios. The diff algorithm described here computes diff in terms of what packages/dependencies/files were added to built artifacts, how the configuration for artifacts changed and what metadata related to the artifact was changed.

The client uses the diff algorithm to compute the change set between the base version (the version that the client has made changes upon) and the local version with the changes. For computing the diff, all the computations in the algorithm are carried out by the client. Both the versions required for computing the diff are present locally on the client side.

Based on the type of artifact, the diff computed is different for different artifacts.

- In case of container image, the diff is computed in terms of packages, file system, image history and image manifest.
- In case of Helm charts, the diff is computed in terms of chart template files, manifest file, values file and requirements file
- In case of singularity images, the diff is computed in terms of a recipe file which represents the built singularity container image. This is a binary file.
- In case of OPA bundle, the diff is computed in terms of manifest file, roles and permission files.

The algorithm that is used for computation can be described as follows:

1. Determine the type of artifact from the media type in the manifest. Based on the type of artifact, the diff is computed for the three categories for each artifact.
2. Find the diff in terms of addition/deletion/modification of packages/files/dependencies to the artifact. The diff is computed for packages and dependencies using the following steps:

- Get the list of packages/dependencies for both versions. This list contains the name of the package and the version of the package.
- Compute the set of packages/ dependencies present in the local version but not in the base version. This set represents packages that were added in the local version.
- Compute the set of packages/dependencies present in the base version but not in the local version. This set represents packages that were removed from the local version.
- Compute the set of packages/dependencies whose version numbers differ between base version and local version. This is a set of packages that were modified in the local version.

For file system diff:

- Get the file content for both versions of the artifact. The file contents are listed by directory. The diff is computed by finding a set of files and directories that were added, deleted and modified.
- Compute the set of directories/files present in the local version but not in the base version. This set represents directories/files that were added in the local version.
- Compute the set of directories/files present in the base version but not in the local version. This set represents directories/files that were removed in the local version.
- Compute the set of directories/files that are present in both versions but the contents are different. This is a set of directories/files that were modified in the local version.

3. Find the diff in artifact configuration. This diff computation is done between the configuration files for the artifact. The diff computation for these files is done using the existing diff algorithms for text like Myers or minimal diff algorithms [128].

4. Find the diff in artifact metadata

   - Get the size of both versions of the artifact
   - Get the details about publishers, maintainers for the artifact
   - Get the details about the tags associated with the artifact
   - Compute the set of tags added or deleted between the two versions of the artifact

5. Display the diff for the user in terms of changes in packages/dependencies/files, changes in configuration, changes in metadata.

4.8 Discussion

In this section, we discuss how the challenges presented by present Harbor architecture and described in Section 4.4 were addressed while designing the proposed VCS. Some of those challenges like concurrency of operations were addressed by leveraging the existing mechanisms implemented by Harbor. While others like persistence of older version were addressed by making few changes in the existing features for garbage collection. The challenges regarding the component structures of artifacts and security of data are addressed by introducing a new internal data structure as described in Section 4.7.1. We also compare the security of our solution with related work.

Concurrency. In order to ensure that the commit and update operations are concurrent and to guarantee atomicity and consistency of operations and the database, we make use of the already implemented feature in Harbor registry. At present, Harbor makes use of Optimistic Concurrency control to maintain concurrency while pushing and pulling artifacts to the registry. We ensure that the commit and update protocols proposed here, follow the same concept of concurrency control while writing data to the shared database or reading data from the shared database.

Garbage Collection. The garbage collection mechanism that Harbor uses, removes all entries from the database that are marked for deletion. The entries are marked for deletion when a delete operation is initiated for either an artifact or a project. For ensuring the persistence of older versions and to provide data about older versions when required, it is necessary that when delete operations are initiated, the commit blobs for those artifacts are not marked for deletion. The artifact blobs are also marked for deletion only if they are not referenced by any other artifact manifest in the repository. This ensures that older versions of artifacts are archived and can be retrieved when needed.

Binary Artifact. The challenge of treating artifacts as structured objects comprising of components like file systems, binary packages and metadata has been addressed by proposing a component structure for artifacts in Section 4.7.4 and a diff algorithm in Section 4.7.5. This enables computation of difference between two artifacts in terms of this structure instead of a string of bytes. Thus, the end user is more informed about how the artifact changes between different versions.

Security of Versions. In order to ensure the authenticity and integrity of data for version control system, we introduce signatures over the commit blob. Having a signature computed over the commit blob will ensure that the current commit has not been tampered with. In addition, because the commit blob contains a pointer to previous commit blob for an artifact, the signature also ensures that no other commit in the artifact repository can be tampered with once the commit operation has been executed successfully. If tampering is attempted, the signature over the blob won't match and tampering will be detected. This ensures the end user that they can trust the authenticity and integrity of the data that has been committed to the version control system.

Comparison with Related Work. The security features in the VCS we propose for OCI artifacts seek to ensure the integrity and authenticity of the data stored at the VCS server, under the assumption that the data committers are able to protect their signing keys. As opposed to this, a solution such as The Update Framework (TUF) [4] seeks to provide protection even against attackers that compromise the signing keys. TUF can potentially be used on top of the VCS we propose, for enhanced security against key compromise.

4.9 Conclusion

In this chapter, we propose a version control system for software deployment artifacts. The system proposed here, when in place, allows the users to determine the changes

between two versions of binary files, provides history of changes and allows rollback to previous versions while not affecting any of the existing features of the artifact registries.

We are the first to propose treating artifacts as structured objects and not binary objects. This allows the system we designed to compute the difference between two artifacts in terms of structural components rather than byte strings unlike existing artifact management systems. This allows the end users to make informed decision about accepting changes in the version and updating to newer versions. We also propose the commit staging protocol that can be used by maintainers to stage their local changes and then publish them to the project repository. Similarly, we propose an update protocol that allows the users to pull changes from remote repository to their local machine.

With the version control system in place, it becomes easier to maintain and keep track of projects using multiple artifacts with multiple versions. Also, since the proposed design does not depend on structure of any particular artifact, the design can be scaled to include newer class of artifacts to the projects easily.

CHAPTER 5

CONCLUSION

The software supply chain consists of various steps such as managing the source code, testing, building, and packaging it into a final product, and distributing the product to end users. During the development of software, the developers use software repositories for different purposes such as source code management (Git, SVN, mercurial), software distribution (PyPI, RubyGems, NPM), or repositories for artifact management (DockerHub, Harbor, JFrog). Recently, these repositories have increasingly been the target of various attacks which have affected a large number of users. Due to this reason, the repositories need mechanisms to ensure the integrity and authenticity of the data. Ensuring the security of data stored and managed using software repositories is a vast and complex topic. Through this work, we have identified several attack surfaces and designed mechanisms to prevent, mitigate or detect those attacks.

In the first part of the work, we introduce a commit signing mechanism that improves the security of centralized version control systems. With this mechanism in place, we ensure integrity, authenticity, and non-repudiation of data committed by developers. We propose an approach for commit signing in conjunction with supporting centralized VCS features such as working with a subset of the repository and allowing clients to work on disjoint sets of files without retrieving each other's changes. The commit signing with support for centralized VCS features is achieved efficiently by signing a Merkle Hash Tree (MHT) computed over the entire repository. We implemented the secure protocol on top of the existing SVN codebase and evaluated its performance with a diverse set of repositories.

In the second part of this work, we presented an approach for certifying the validity of software projects hosted on community repositories. We have introduced a Software Certification Service (SCS), which allows software publishers to prove ownership of their projects and then get a project certificate that binds the project owner and the project name to a public key. Although inspired by the ACME protocol, the proposed certification protocol is fundamentally different in its attack mitigation capabilities and in how ownership is proven. It can also be fully automated on the SCS side. We also deployed the SCS service to several community repositories, including PyPI, RubyGems, and NPM, to automate the issuance of certificates for projects hosted on these repositories. By design, the SCS service does not require any changes to these community repositories, making it deployable immediately and serving as an incentive for adoption. As part of future work, we plan to extend the SCS service to more community repositories (at present, we support community repositories where each project has a dedicated webpage containing a project description section) and explore other use cases that can benefit from automated verification.

In the last part of this work, we focus on the registries that store and manage project deployment artifacts such as container images, Helm charts, OPA bundles, and Singularity Images. These repositories are used for storing, managing and deploying software projects. With the increase in popularity of containers and related artifacts, it is necessary to have uniform standards for storage, management and distribution of these artifacts. In the course of this work, we present uniform standards for version control in containers and related artifacts. We have identified challenges in designing a uniform version control system and proposed a solution addressing these challenges. We propose a uniform standard for version control that is independent of the type of artifact being version controlled. We design the commit and update protocols that allow the user to version control project artifacts. We also proposed a component structure for these artifacts. This component structure can then be leveraged to

determine changes between different versions of artifacts. As part of future work, we plan to examine the performance overheads of the proposed solution. We also plan to explore how the proposed solution fares when used in conjunction with other widely used management and orchestration solutions.

BIBLIOGRAPHY

[1] S. Vaidya, S. Torres-Arias, R. Curtmola, and J. Cappos, "Commit signatures for centralized version control systems," in *Proc. of ICT Systems Security and Privacy Protection*. Springer International Publishing, 2019, pp. 359–373.

[2] "Let's encrypt-free ssl/tls certificates," https://letsencrypt.org/, [Online; accessed 2022-6-30].

[3] "/specs/tuf-spec.txt - TUF: The Update Framework," https://www.updateframework.com/browser/specs/tuf-spec.txt, [Online; accessed 2022-6-30].

[4] "TUF: The Update Framework," https://www.updateframework.com/, [Online; accessed 2022-6-30].

[5] T. K. Kuppusamy, S. Torres-Arias, V. Diaz, and J. Cappos, "Diplomat: Using delegations to protect community repositories," in *In Proc. of 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 567–581.

[6] "What are docker images?" https://www.docker.com/resources/what-container/, [Online; accessed 2022-6-30].

[7] "What are helm charts?" https://helm.sh/, [Online; accessed 2022-6-30].

[8] "What are opa bundles?" https://www.openpolicyagent.org/, [Online; accessed 2022-6-30].

[9] "What are singularity images?" https://sylabs.io/, [Online; accessed 2022-6-30].

[10] "Git," https://git-scm.com/, [Online; accessed 2022-6-30].

[11] "Apache subversion," https://subversion.apache.org/, [Online; accessed 2022-6-30].

[12] "Mercurial," https://www.mercurial-scm.org/, [Online; accessed 2022-6-30].

[13] "Concurrent versions system," https://www.nongnu.org/cvs/, [Online; accessed 2022-6-30].

[14] "Github," https://github.com/, [Online; accessed 2022-6-30].

[15] "Gitlab," https://about.gitlab.com/, [Online; accessed 2022-6-30].

[16] "Sourceforge," https://sourceforge.net/, [Online; accessed 2022-6-30].

[17] "Internet security threat report, symantec," https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf, [Online; accessed 2018-8-23].

[18] "Adobe source code breach; it's bad, real bad," https://gigaom.com/2013/10/04/adobe-source-code-breech-its-bad-real-bad/, [Online; accessed 2019-6-30].

[19] Talos, "Ccleanup: A vast number of machines at risk," https://blog.talosintelligence.com/2017/09/avast-distributes-malware.html, [Online; accessed 2017-9-30].

[20] "Kernel.org linux repository rooted in hack attack," http://www.theregister.co.uk/2011/08/31/linux_kernel_security_breach/, [Online; accessed 2018-7-31].

[21] "Bitcoin gold critical warning," https://bitcoingold.org/critical-warning-nov-26/, [Online; accessed 2022-6-30].

[22] "Cloud source host Code Spaces hacked, developers lose code," https://www.gamasutra.com/view/news/219462/Cloud_source_host_Code_Spaces_hacked_developers_lose_code.php, [Online; accessed 2022-6-30].

[23] "Breaching Fort Apache.org - What went wrong?" http://www.theregister.co.uk/2009/09/03/apache_website_breach_postmortem/, [Online; accessed 2022-6-30].

[24] "'Google' Hackers Had Ability to Alter Source Code," https://www.wired.com/2010/03/source-code-hacks/, [Online; accessed 2022-6-30].

[25] "The Linux Backdoor Attempt of 2003," https://freedom-to-tinker.com/2013/10/09/the-linux-backdoor-attempt-of-2003/, [Online; accessed 2022-6-30].

[26] "Compare Repositories," https://www.openhub.net/repositories/compare, [Online; accessed 2022-6-30].

[27] "Bazaar," http://bazaar.canonical.com/en/, [Online; accessed 2022-6-30].

[28] "Perforce Helix Core," https://www.perforce.com/products/helix-core, [Online; accessed 2022-6-30].

[29] "Surround SCM," https://www.perforce.com/products/surround-scm, [Online; accessed 2022-6-30].

[30] "StarTeam," https://www.microfocus.com/products/change-management/starteam/, [Online; accessed 2022-6-30].

[31] "Vault," http://www.sourcegear.com/vault/, [Online; accessed 2022-6-30].

[32] R. Merkle, "Protocols for public key cryptosystems," in *Proc. of IEEE symposium on security and privacy*, 1980.

[33] "Git internal objects," https://git-scm.com/book/en/v2/Git-Internals-Git-Objects, accessed: 2021-02-18.

[34] D. Mazieres and D. Shasha, "Building secure file systems ' out of byzantine storage," in *Proc. of the 21st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (SPDC '02)*, 2002, pp. 108–117.

[35] J. Li, M. Krohn, D. Mazieres, and D. Shasha, "Secure untrusted data repository (sundr)," in *Proc. of the 6th USENIX Symposium on Operating Systems Design & Implementation (OSDI '04)*, 2004.

[36] S. Torres-Arias, A. K. Ammula, R. Curtmola, and J. Cappos, "On omitting commits and committing omissions: Preventing git metadata tampering that (re)introduces software vulnerabilities)," in *Proc. of the 25th USENIX Security Symposium*, 2016.

[37] *China, GitHub and the man-in-the-middle*, [Online; accessed 2019-5-18].

[38] arsTechnica, *Meet "Great Cannon", the man-in-the-middle weapon China used on GitHub*, arstechnica.com/security/2015/04/meet-great-cannon-the-man-in-the-middle-weapon-china-used-on-github/, [Online; accessed 2022-6-30].

[39] B. Marczak, N. Weaver, J. Dalek, R. Ensafi, D. Fifield, S. McKune, A. Rey, J. Scott-Railton, R. Deibert, and V. Paxson, "An analysis of china's "great cannon"," in *Proc. of Fifth USENIX FOCI*, 2015.

[40] C. Soghoian and S. Stamm, "Certified lies: Detecting and defeating government interception attacks against ssl (short paper)," in *Proc. of The 16th International Conf. on Financial Cryptography and Data Security (FC '12)*, 2012.

[41] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. Bailey, J. A. Halderman, and V. Paxson, "The security impact of HTTPS interception," in *Proc. of Netw. and Distr. Sys. Security Symp. (NDSS)*, 2017.

[42] B. Moller, T. Duong, and K. Kotowicz, *This POODLE bites: exploiting the SSL 3.0 fallback*, https://www.openssl.org/~bodo/ssl-poodle.pdf, [Online; accessed 2022-6-30].

[43] S. Stricot-Tarboton, S. Chaisiri, and R. K. L. Ko, "Taxonomy of man-in-the-middle attacks on https," in *Proc. of 2016 IEEE Trustcom/BigDataSE/ISPA*, 2016, pp. 527–534.

[44] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohney, S. Engels, C. Paar, and Y. Shavitt, "DROWN: Breaking TLS using sslv2," in *Proc. of 25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 689–706.

[45] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman, "Analysis of the HTTPS certificate ecosystem," in *Proc. of the 2013 conference on Internet measurement conference (IMC)*, 2013, pp. 291–304.

[46] "Filezilla," https://filezilla-project.org/, [Online; accessed 2019-2-15].

[47] "Gcc," https://gcc.gnu.org/, [Online; accessed 2019-3-30].

[48] "Svn changeset signing," http://svn.apache.org/repos/asf/subversion/trunk/notes/changeset-signing.txt, [Online; accessed 2019-4-23].

[49] "Gnu bazaar gnupg signatures," http://doc.bazaar.canonical.com/beta/en/user-guide/gpg_signatures.html, [Online; accessed 2018-6-30].

[50] D. A. Wheeler, "Software configuration management (scm) security," https://www.dwheeler.com/essays/scm-security.html, [Online; accessed 2022-6-30].

[51] "Git commit signature," https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work, [Online; accessed 2022-6-30].

[52] M. Gerwitz, "A git horror story: Repository integrity with signed commits," https://mikegerwitz.com/papers/git-horror-story, [Online; accessed 2022-6-30].

[53] B. Chen and R. Curtmola, "Auditable version control systems," in *Proc. of The 21st ISOC Annual Network & Distributed System Security Symposium*, February 2014.

[54] "Vfs for git," https://vfsforgit.org/, [Online; accessed 2022-6-30].

[55] "Teach git to support a virtual (partially populated) work directory," https://public-inbox.org/git/20181213194107.31572-1-peartben@gmail.com/, [Online; accessed 2022-6-30].

[56] "Python packaging index," https://pypi.org, [Online; accessed 2022-6-30].

[57] "Rubygems statistics," https://rubygems.org/stats, [Online; accessed 2022-6-30].

[58] "Javascript package manager," https://npmjs.com, [Online; accessed 2022-6-30].

[59] "Pypi download stats," https://pypistats.org/packages/__all__, [Online; accessed 2022-6-30].

[60] "Npm download stats," https://npmcharts.com/, [Online; accessed 2022-6-30].

[61] D. Goodin, "Software downloaded 30,000 times from PyPI ransacked developers' machines," arstechnica.com/gadgets/2021/07/malicious-pypi-packages-caught-stealing-developer-data-and-injecting-code/, July 2021.

[62] J. Burt, "Supply Chain Flaws Found in Python Package Repository," esecurityplanet.com/threats/supply-chain-flaws-found-in-python-package-repository/, August 2021.

[63] A. Polkovnychenko and S. Menashe, "Python Malware Imitates Signed PyPI Traffic in Novel Exfiltration Technique," jfrog.com/blog/python-malware-imitates-signed-pypi-traffic-in-novel-exfiltration-technique/, November 2021.

[64] A. Sharma, "Sonatype Catches New PyPI Cryptomining Malware," blog.sonatype.com/sonatype-catches-new-pypi-cryptomining-malware-via-automated-detection/, June 2021.

[65] J. Ruohonen, K. Hjerppe, and K. Rindell, "A large-scale security-oriented static analysis of python packages in pypi," *CoRR*, vol. abs/2107.12699, 2021. [Online]. Available: https://arxiv.org/abs/2107.12699

[66] "Supply-chain attack hits rubygems repository with 725 malicious packages," arstechnica.com/information-technology/2020/04/725-bitcoin-stealing-apps-snuck-into-ruby-repository/, 2020, [Online; accessed 2022-6-30].

[67] "Malicious npm packages target amazon, slack with new dependency attacks," bleepingcomputer.com/news/security/malicious-npm-packages-target-amazon-slack-with-new-dependency-attacks/, 2021, [Online; accessed 2022-6-30].

[68] A. Barsan, "Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies," https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610/, February 2021.

[69] R. Lakshmanan, "Two NPM Packages With 22 Million Weekly Downloads Found Backdoored," https://thehackernews.com/2021/11/two-npm-packages-with-22-million-weekly.html, [Online; accessed 2022-6-30].

[70] J. Aguirre, "Fake npm Roblox API Package Installs Ransomware and has a Spooky Surprise," https://blog.sonatype.com/fake-npm-roblox-api-package-installs-ransomware-spooky-surprise, [Online; accessed 2022-6-30].

[71] C. Cimpanu, "Malware found in npm package with millions of weekly downloads," https://therecord.media/malware-found-in-npm-package-with-millions-of-weekly-downloads/, [Online; accessed 2022-6-30].

[72] L. Abrams, "Malicious NPM packages target Amazon, Slack with new dependency attacks," bleepingcomputer.com/news/security/malicious-npm-packages-target-amazon-slack-with-new-dependency-attacks/, [Online; accessed 2022-6-30].

[73] "Leading certificate authorities and microsoft introduce new standards to protect consumers online," https://tinyurl.com/2wahhcfe, [Online; accessed 2022-6-30].

[74] "Minimum requirements for the issuance and management of publicly-trusted code signing certificates,," casecurity.org/wp-content/uploads/2016/09/Minimum-requirements-for-the-Issuance-and-Management-of-code-signing.pdf, [Online; accessed 2022-6-30].

[75] "Introduction to code signing," docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/ms537361(v=vs.85), [Online; accessed 2022-6-30].

[76] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kasten, "Automatic Certificate Management Environment (ACME)," RFC 8555, [Online; accessed 2022-6-30]. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8555

[77] "Let's encrypt stats," https://letsencrypt.org/stats/, [Online; accessed 2022-6-30].

[78] "Keybase," https://https://keybase.io/, [Online; accessed 2022-6-30].

[79] "Security/Binary Transparency," https://wiki.mozilla.org/Security/Binary_Transparency, [Online; accessed 2022-6-30].

[80] "[meta] Binary Transparency on Firefox," https://bugzilla.mozilla.org/show_bug.cgi?id=1341395, [Online; accessed 2022-6-30].

[81] "add signed tree head decoding to certificate transparency implementation," https://bugzilla.mozilla.org/show_bug.cgi?id=1503348, [Online; accessed 2022-6-30].

[82] "crt.sh: identity search," https://crt.sh/?q=fx-trans.net&dir=v&sort=1&group=none, [Online; accessed 2022-6-30].

[83] "Comprehensive Perl Archive Network," https://www.cpan.org/, accessed: 2021-5-24.

[84] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 8259, Dec. 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8259

[85] "Bitcoin gold issues critical alert," enterprisetimes.co.uk/2017/11/27/bitcoin-gold-issues-critical-alert, [Online; accessed 2022-6-30].

[86] "Npm packages disguised as roblox api code caught carrying ransomware," https://www.theregister.com/2021/10/27/npm_roblox_ransomware/, [Online; accessed 2022-6-30].

[87] "Typosquatting attacks on rubygems," thehackernews.com/2020/04/rubygem-typosquatting-malware.html, [Online; accessed 2022-6-30].

[88] "Acme server boulder," https://github.com/letsencrypt/boulder, [Online; accessed 2022-6-30].

[89] "Acme client implementation," https://letsencrypt.org/docs/client-options/, [Online; accessed 2022-6-30].

[90] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, "Package Management Security," University of Arizona, Tech. Rep., [Online; accessed 2022-6-30].

[91] ——, "A look in the mirror: Attacks on package managers," in *Proc. of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 565–574. [Online]. Available: https://doi.org/10.1145/1455770.1455841

[92] T. K. Kuppusamy, V. Diaz, and J. Cappos, "Mercury: Bandwidth-effective prevention of rollback attacks against community repositories," in *Proc. of the 2017 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '17, 2017, p. 673–688.

[93] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos, "In-toto: Providing farm-to-table guarantees for bits and bytes," in *Proc. of the 28th USENIX Conference on Security Symposium*, ser. SEC'19, 2019, p. 1393–1410.

[94] R. K. Vaidya, L. D. Carli, D. Davidson, and V. Rastogi, "Security issues in language-based sofware ecosystems," *CoRR*, vol. abs/1903.02613, 2019.

[95] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *Proc. of 28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 995–1010.

[96] K. Garrett, G. Ferreira, L. Jia, J. Sunshine, and C. Kästner, "Detecting suspicious package updates," in *Proc. of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER '19. IEEE Press, 2019, p. 13–16.

[97] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proc. of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 181–191.

[98] D.-L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Typosquatting and combosquatting attacks on the python ecosystem," in *Proc. of 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, 2020, pp. 509–514.

[99] H. Finney, L. Donnerhacke, J. Callas, R. L. Thayer, and D. Shaw, "OpenPGP Message Format," RFC 4880, [Online; accessed 2022-6-30].

[100] "Gpg signatures in pypi," https://github.com/pypa/twine/issues/157, accessed: 2015-6-24.

[101] "Npm pgp machinery," https://blog.npmjs.org/post/172999548390/new-pgp-machinery.html, accessed: 2022-6-30.

[102] "Rubygems security," https://guides.rubygems.org/security/, [Online; accessed 2022-6-30].

[103] "Docker index statistics," https://tinyurl.com/ysvernua, [Online; accessed 2022-6-30].

[104] "Docker hub," https://hub.docker.com/, [Online; accessed 2022-6-30].

[105] "Artifact hub," https://artifacthub.io/, [Online; accessed 2022-6-30].

[106] "Open container initiative," https://opencontainers.org/, [Online; accessed 2022-6-30].

[107] "Jfrog artifactory," https://jfrog.com/, [Online; accessed 2022-6-30].

[108] "Harbor registry," https://goharbor.io/, [Online; accessed 2022-6-30].

[109] "Oci registry as service," https://oras.land/, [Online; accessed 2022-6-30].

[110] "Oci artifacts," https://github.com/opencontainers/artifacts, [Online; accessed 2022-6-30].

[111] "Docker images," https://jfrog.com/knowledge-base/a-beginners-guide-to-understanding-and-building-docker-images/, accessed: 2022-06-22.

[112] "Introduction to helm charts," https://www.bmc.com/blogs/kubernetes-helm-charts/, accessed: 2021-09-23.

[113] "Kubernetes," https://kubernetes.io/, accessed: 2022-06-23.

[114] "Opa bundle format," https://www.openpolicyagent.org/docs/v0.12.2/, accessed: 2022-05-12.

[115] "Cncf landscape," https://github.com/cncf/landscape#trail-map, accessed: 2022-02-21.

[116] "Singularity hub," https://singularity-hub.org/, accessed: 2022-06-23.

[117] "Overcoming docker tag mutability," https://www.whitesourcesoftware.com/free-developer-tools/blog/overcoming-dockers-mutable-image-tags/, accessed: 2022-06-23.

[118] Wikipedia contributors, "Optimistic concurrency control — Wikipedia, the free encyclopedia," [Online; accessed 2022-6-30]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Optimistic_concurrency_control&oldid=1091159570

[119] "Garbage collection," https://goharbor.io/docs/1.10/administration/garbage-collection/, accessed: 2022-6-30.

[120] "Cloud repo," https://www.cloudrepo.io/, accessed: 2022-06-23.

[121] "Git large file system," https://git-lfs.github.com/, accessed: 2022-06-23.

[122] "Boar binary repository," https://github.com/mekberg/boar, accessed: 2022-06-23.

[123] "Attack of the mutant tags," https://sysdig.com/blog/toctou-tag-mutability/, accessed: 2021-11-28.

[124] "Getting started with yarn," https://classic.yarnpkg.com/en/docs/getting-started, accessed: 2022-06-23.

[125] "Lighttag wrong tag postmortem," https://www.lighttag.io/blog/postmortem-docker-swarm-wrong-tag/, accessed: 2022-06-23.

[126] "Postgresql: World's most advanced open source database," https://www.postgresql.org/, accessed: 2022-7-2.

[127] "Harbor architecture," https://github.com/goharbor/harbor/wiki/Architecture-Overview-of-Harbor, accessed: 2022-06-23.

[128] "Diff algorithms," https://blog.jcoglan.com/2017/02/12/the-myers-diff-algorithm-part-1/, accessed: 2022-06-23.