Commit Signatures for Centralized Version Control Systems

Sangat Vaidya[1], Santiago Torres-Arias[2], Reza Curtmola[1(✉)], and Justin Cappos[2]

[1] New Jersey Institute of Technology, Newark, NJ, USA
reza.curtmola@njit.edu

[2] Tandon School of Engineering, New York University, New York, NY, USA

Abstract. 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.

We design the first commit signing mechanism for centralized version control systems, which supports 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. 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. With our solution in place, the VCS security model is substantially improved.

Keywords: SVN · Commit signature · Version control system

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 [2], Mercurial [18], and CVS [7].

The full version of this paper is available as a technical report [34].

Published by Springer Nature Switzerland AG 2019
G. Dhillon et al. (Eds.): SEC 2019, IFIP AICT 562, pp. 359–373, 2019.
https://doi.org/10.1007/978-3-030-22312-0_25

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 [12], GitLab [13], or Sourceforge [20]. 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 [16], and have affected most types of repositories, including Git [1,4,17,32], Subversion [5,6], Perforce [15], and CVS [26].

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. Perhaps surprisingly, several other VCS-es, such as Apache Subversion [2] (known as SVN), lack this ability and are vulnerable to attacks that manipulate files on a remote repository in an undetectable fashion.

Contributions. In this work, 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.

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. We describe a solution for SVN, but our techniques are applicable to other VCS-es that fit this model, such as GNU Bazaar [3], Perforce Helix Core [19], Surround SCM [22], StarTeam [21], and Vault [27].
– 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 Background

This section provides background on version control systems (VCS-es) that have a (centralized) client-server architecture [3,19,21,22,27] 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 [2], an open source VCS that is representative for this class of VCS-es.

2.1 Centralized Version Control Systems

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*. 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.

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 LWC with the changes made by other clients, she uses the "`update`" command. The codebase revisions are referred to by a unique identifier called a *revision number*. In SVN, this is

PROTOCOL: Commit

1: **for** (each file F in the commit changeset) **do**
2: $C \rightarrow S : \delta$ // Client computes and sends δ, 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

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 it local revision number for F
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

an integer number that has value 1 initially and is incremented by 1 every time a client commits changes to the repository.

The server stores revisions using *skip delta encoding*, in which only the first revision is stored in its entirety and each subsequent revision is stored as the difference (*i.e.*, delta) relative to an earlier revision [24].

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 $F_i = F_j + \delta$ 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 is 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 Merkle Hash Trees

A Merkle Hash Tree (MHT) [31] 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 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. 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 to be verified and the root node.

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

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

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. 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 allows clients 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 the most up-to-date version for the entire repository before pushing commits to the main repository (*i.e.*, they need to retrieve all changes made anywhere in the repository before pushing changes). This ensures that a Git client has updated metadata about the entire repository before pushing 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.

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.

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 arbitrarily tamper with data. 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 desirable defense goals.

Tampering Attack. The attacker can arbitrarily tamper with the repository data, such as 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. 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. A defense should securely bind together the files versions that belong to a revision, and should also bind them to the revision identifier.

4.2 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 protects against 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.

5 Commit Signatures for Centralized VCS-es

We now present our design for enabling commit signatures by enhancing the standard **Commit** and **Update** protocols. We use the following notation, in addition to what we defined in Sect. 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 $Sign$ and $Verify$ use the private and public keys of the client who committed the revision. Due to space limitations, the security analysis of these protocols is included in the full version of the paper.

Secure Commit Protocol. We now present the **Secure_Commit** protocol. The client has a commit changeset with changes on top of revision $i-1$, and wants to commit revision i. 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

PROTOCOL: Secure_Commit

1: // Steps 1-6 are the same as in the standard **Commit** protocol
7: 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
8: $S \rightarrow C : i, P_{i-1}, CSIG_{i-1}, RevInfo_{i-1}$ // S sends the new revision number i, the proof for the changeset, and the commit signature and revision information for revision $i-1$
9: **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
10: C computes the $MHTROOT_i$ using P_{i-1} and revision i of the files in the commit changeset
11: C sets $RevInfo_i = i, i-1, ID_{client}$
12: C computes $CSIG_i = Sign(MHTROOT_i, RevInfo_i)$
13: $C \rightarrow S : CSIG_i, RevInfo_i$
14: S computes the MHT for revision i using the MHT for revision $i-1$ and the client's changeset
15: S stores $CSIG_{i-1}, RevInfo_i$ and the MHT for revision i

PROTOCOL: Secure_Update

1: // Steps 1-7 are the same as in the standard **Update** protocol
8: S computes proof P_i // S uses revision i of the repository to compute proof P_i relative to the client's update set
9: $S \rightarrow C : P_i, CSIG_i, RevInfo_i$ // S sends the proof for the update set, and the commit signature and revision information for revision i
10: **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
11: **for** (each file F in the update set) **do**
12: C stores F_i in its local repository

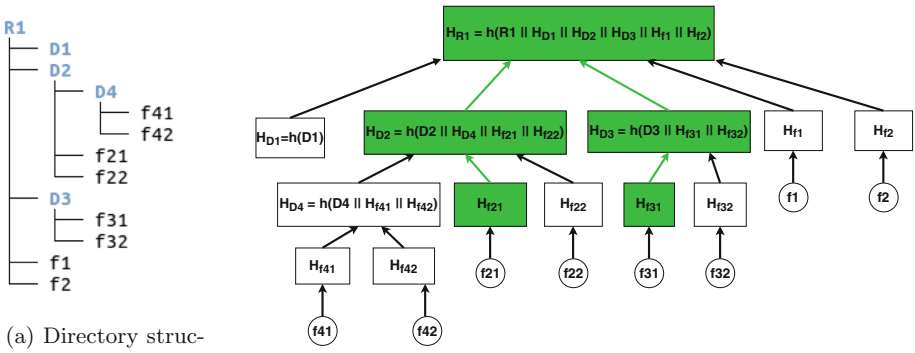
7). We describe how this proof is computed and verified in Sect. 5.1. After receiving the new revision number, the proof, and the commit signature and revision information for revision $i-1$ (line 8), the client verifies the validity of the proof (line 9). The client then uses this proof and the files 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).

Secure Update Protocol. The client wants to retrieve revision i for a set of files in the repository, referred to as the *update set*. 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 (line 9). The client then verifies this proof (line 10). After finalizing the update, the client sets i as revision number for all the files in the update set.

5.1 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 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 a revision of the repository. The MHT leaves are hashes of files, which 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 1 shows the directory structure and the corresponding MHT for a revision of repository R1.



(a) Directory structure of repository R1

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

Fig. 1. 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 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, consider the MHT for the repository R1 as shown in Fig. 1b. 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 consists 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 Fig. 1b using solid-filled nodes. Thus, the proof is $\{H_{D4}, H_{f22}, H_{f32}, H_{D1}, H_{f1}, H_{f2}\}$.

6 Implementation and Experimental Evaluation

6.1 Implementation and Experimental Setup

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.

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), 16 GB RAM, and a 500 GB hard disk with ext4 file system. The system runs Ubuntu 16.04 LTS, kernel v. 4.10.14-041014-generic, and OpenSSL 1.0.2g.

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 [8], SVN [2], and GCC [9], as shown in Table 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 included in our experiments the overhead average over the last 100 revisions of GCC (labeled GCC2). All the data points in the experimental evaluation section are averaged over three independent runs.

6.2 Experimental Evaluation for Commit Operations

End-to-End Delay. The results for end-to-end delay per commit operation are shown in Table 3. 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 s 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 s.

Communication Overhead. Table 2 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. SSVN adds between 0.27 KB to 0.8 KB of communication overhead from server to client. This overhead is caused by 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. There is no storage overhead on the client side as the client does not store any additional data in SSVN. On the server side, Table 4 shows that SSVN adds between 0.1 MB–0.16 MB per commit over SVN for FileZilla,

Table 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	21 KB	18 KB	6 KB
Repository size (all revisions)	29.2 MB	43.9 MB	492.7 MB

Table 2. Network communication for committing one revision (in KBs): from client to server (top two rows), from server to client (bottom two rows).

	FileZilla	SVN	GCC1	GCC2
SVN	35.565	46.672	4.676	20.347
SSVN	35.825	46.934	4.933	20.605
SVN	0.865	1.095	0.539	2.476
SSVN	1.137	1.432	0.962	3.275

Table 3. Commit time per 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

Table 4. Server storage per 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

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, because 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 3 MB.

6.3 Experimental Evaluation for Update Operations

End-to-End Delay. The results for end-to-end delay per update operation are shown in Table 5. 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 s. Even for GCC2, the maximum increase remains modest, at 0.638 s. 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.

Communication Overhead. Table 6 shows that SSVN adds between 0.24 KB–0.66 KB to the communication from the server to the client. This overhead is

Table 5. Update time per 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 6. Network communication for updating one revision (in KBs): from client to server (top two rows), from server to client (bottom two rows).

	FileZilla	SVN	GCC1	GCC2
SVN	1.243	1.328	0.953	10.235
SSVN	1.235	1.548	1.045	11.369
SVN	36.342	49.978	5.782	54.678
SSVN	36.745	50.225	6.245	55.346

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.

7 Related Work

Even though an early proposal draft for SVN changeset signing has been considered [23], 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 [3] is a centralized VCS that allows to sign and verify commits [14] 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 [35] 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 work, 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 [11]. Metadata manipulation attacks against Git were identified by Torres-Arias *et al.* [33]. Gerwitz [30] 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 paper (Sect. 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 [29] 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.

8 Conclusion

In this work, 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 [25,28].

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 0.5 KB network communication and less than 0.2 s end-to-end delay per commit/update; even for very large repositories, the communication overhead is under 1 KB and end-to-end delay overhead remains under 1 s per commit/update.

Acknowledgments. This research was supported by the NSF under Grants No. CNS 1801430 and DGE 1565478. We would like to thank Ruchir Arya for contributions to an earlier version of this work.

References

1. Adobe source code breach; it's bad, real bad. https://gigaom.com/2013/10/04/adobe-source-code-breech-its-bad-real-bad/
2. Apache subversion. https://subversion.apache.org/
3. Bazaar. http://bazaar.canonical.com/en/
4. Bitcoin gold critical warning. https://bitcoingold.org/critical-warning-nov-26/
5. Breaching Fort Apache.org - What went wrong?. http://www.theregister.co.uk/2009/09/03/apache_website_breach_postmortem/

6. 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
7. Concurrent versions system. https://www.nongnu.org/cvs/
8. Filezilla. https://filezilla-project.org/
9. GCC. https://gcc.gnu.org/
10. Git. https://git-scm.com/
11. Git commit signature. https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work
12. GitHub. https://github.com/
13. GitLab. https://about.gitlab.com/
14. Gnu bazaar GnuPG signatures. http://doc.bazaar.canonical.com/beta/en/user-guide/gpg_signatures.html
15. 'Google' Hackers Had Ability to Alter Source Code. https://www.wired.com/2010/03/source-code-hacks/
16. Internet security threat report, symantec. https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf
17. Kernel.org linux repository rooted in hack attack. http://www.theregister.co.uk/2011/08/31/linux_kernel_security_breach/
18. Mercurial. https://www.mercurial-scm.org/
19. Perforce Helix Core. https://www.perforce.com/products/helix-core
20. Sourceforge. https://sourceforge.net/
21. StarTeam. https://www.microfocus.com/products/change-management/starteam/
22. Surround SCM. https://www.perforce.com/products/surround-scm
23. SVN changeset signing. http://svn.apache.org/repos/asf/subversion/trunk/notes/changeset-signing.txt
24. SVN skip deltas. http://svn.apache.org/repos/asf/subversion/trunk/notes/skip-deltas
25. Teach git to support a virtual (partially populated) work directory. https://public-inbox.org/git/20181213194107.31572-1-peartben@gmail.com/
26. The Linux Backdoor Attempt of 2003. https://freedom-to-tinker.com/2013/10/09/the-linux-backdoor-attempt-of-2003/
27. Vault. http://www.sourcegear.com/vault/
28. VFS for Git. https://vfsforgit.org/
29. Chen, B., Curtmola, R.: Auditable version control systems. In: Proceedings of the 21st ISOC Annual Network & Distributed System Security Symposium, February 2014
30. Gerwitz, M.: A git horror story: repository integrity with signed commits. https://mikegerwitz.com/papers/git-horror-story
31. Merkle, R.: Protocols for public key cryptosystems. In: Proceedings of IEEE Symposium on Security and Privacy (1980)
32. Talos: CCleanup: a vast number of machines at risk. https://blog.talosintelligence.com/2017/09/avast-distributes-malware.html
33. Torres-Arias, S., Ammula, A.K., Curtmola, R., Cappos, J.: On omitting commits and committing omissions: preventing git metadata tampering that (re)introduces software vulnerabilities. In: Proceedings of the 25th USENIX Security Symposium (2016)
34. Vaidya, S., Torres-Arias, S., Curtmola, R., Cappos, J.: Commit signatures for centralized version control systems. Technical report, NJIT, March 2019
35. Wheeler, D.A.: Software configuration management (SCM) security. https://www.dwheeler.com/essays/scm-security.html