Integrity Protection for Revision Control

Christian Cachin[1] and Martin Geisler[2]

[1] IBM Research, Zurich Research Laboratory, Switzerland, cca@zurich.ibm.com
[2] Department of Computer Science, University of Aarhus, Denmark, mg@cs.au.dk

Abstract. Users of online-collaboration tools and network storage services place considerable trust in their providers. This paper presents a novel approach for protecting data integrity in revision control systems hosted by an untrusted provider. It guarantees atomic read and write operations on the shared data when the service is correct and preserves fork-linearizability when the service is faulty. A prototype has been implemented on top of the Subversion revision control system; benchmarks show that the approach is practical.

Keywords. Hash trees, memory checking, fork linearizability, storage security, applied cryptography.

1 Introduction

Nowadays people from all continents and all time zones collaborate together in global companies and other organizations, formal or not. Prominent examples are open-source development projects, such as the GNU/Linux operating system. For exchanging documents and storing the output of their work, they typically rely on a remote provider that hosts a shared storage service. An important class of such storage services are *revision control systems* (RCS) that facilitate collaboration on a set of documents that belong together and exist in multiple versions.

Although the collaborators trust the storage provider to preserve their documents, there are good reasons to verify that the provider indeed behaves correctly. For example, there are reported cases of break-ins to popular open-source repositories, where security-critical operating system code may have been altered undetectedly [7]. In cooperations that span multiple organizations, the storage provider often is a third party with little interest in the resulting work. Generally, verification reduces trust in the storage provider. To protect against faulty or corrupted storage providers, cryptographic protection methods are needed.

In this paper, we address *cryptographic integrity protection* for revision control systems. They represent the most important kind of multi-user storage and collaboration tools today, together with Wikis. We assume that clients are isolated and communicate directly with each other only under special circumstances; in fact, many clients may not even know each other. Our goal is to obtain a strong guarantee that a potentially faulty service provider has not altered the shared data.

The clients may use public-key signatures to authenticate their operations; this ensures that no unauthorized party can forge data in the repository. But in our model, replay attacks by a malicious storage server cannot be prevented, i.e., the server may return an outdated value to a reader, omitting a more recent update by another client. SUNDR [11] was the first storage system to address this problem by providing every client with a *fork-linearizable* view of the shared data. This notion ensures that all operations that a client does see are observed in the agreed linearization order, and if the server causes the views of two clients to differ in a single operation, they may never again see each others operations. This makes even subtle changes to the stored data easily detectable.

In this work, we describe the design and implementation of a consistent revision control system that preserves fork-linearizability. It relies on the fork-linearizable storage protocol of Cachin et al. [5] that reduces the communication overhead by an order of magnitude compared to the protocol of SUNDR. Our implementation extends the popular revision control system Subversion in a modular way.

The challenge in our work lies in the details of the integration of the fork-linearizable storage protocol with a revision control system. First, the abstract storage protocol uses only simple read and write operations on a file, whereas the revision control system implements transactions that usually read and update many files at once. Second, our goal is to be transparent to the server side of the underlying revision control system; therefore, we still rely on it to serialize concurrent updates. The implementation of our consistent revision control system merely extends this serialization order with the cryptographic consistency guarantees. Finally, the cryptographic operations must not be overly expensive; our hash-tree implementation exploits caching of the tree nodes and maintains them in the Subversion repository itself. This adds only little extra storage on top of the unchecked repository and requires few more operations.

1.1 Related Work

Protecting the integrity of stored data is an important question with a long history. But good solutions are needed today more than ever before [2], because personal and institutional data is stored and archived electronically. We describe here only a selection of the literature that uses the same model as our system, i.e., a remote untrusted bulk storage provider that offers read and write operations, accessed by one or more isolated clients with a small trusted memory.

Blum et al. [3] formalize the problem of memory checking and present the classical protection scheme based on hash trees [15]. With a memory consisting of n items and with random-access read and write operations to the memory, hash trees incur an overhead of $O(\log n)$ cryptographic operations. Several storage-system prototypes protect data integrity using hash trees; TDB [12] and SiRiUS [8] are two prominent examples. A similar approach has been proposed for protecting a CPU equipped with a trusted cache against unauthorized modifications to the main memory [6]. For database systems accessed through a query

interface, Mykletun et al. [16] analyze the cost of integrity protection with cryptographic signatures that can be aggregated to reduce the space overhead. The recent work of Papamanthou et al. [17] shows how an array of data items can be authenticated with constant overhead for reading and sub-linear overhead for writing.

All systems mentioned so far consider either only one client or construct an abstraction of the trusted memory between clients (e.g., with digital signatures). The SUNDR system [11] is the only one protecting integrity for storage space shared by multiple clients that do not communicate among themselves. SUNDR guarantees linearizability when the storage service is correct and fork-linearizability when the service is faulty.

In distributed revision control, the two popular systems Git (`http://git.or.cz/`) and Mercurial (`http://www.selenic.com/mercurial/`) both employ hashes for identifying revisions. Without digital signatures, a corrupted server can trivially present modified changesets to a client (a changeset is the unit of an update between two revisions). The clients will no longer agree on the hashes identifying the revisions, but the server can keep passing content back and forth between the clients. Even if every client would sign all its updates, replay attacks would still be possible despite the use of hashes. Distributed revision control systems explicitly allow offline commits, and so the server can withhold changesets and claim that it has not seen them yet.

In practice, many open-source projects also publish digests or even cryptographic signatures on every release of their code. But since the cryptographic operations for authentication and verification are not transparently integrated with the storage mechanism, they require some manual intervention; hence, this method is not suitable for everyday collaboration.

1.2 Overview of the Paper

The remainder of the paper is organized as follows. Section 2 presents our system model and the design for our consistent revision control system. Section 3 describes our implementation. We evaluated our prototype system and present the results in Section 4. Section 5 discusses some limitations of our system and presents an outlook.

2 Design

This section presents the design of our consistent revision control system. In Section 2.1, we first describe the assumptions used by our system and the properties that it guarantees. We then introduce our abstract consistent storage service in Section 2.2, which provides a fork-linearizable storage space for small values, and review those properties of revision control systems that are relevant for our work in Section 2.3. In Section 2.4, we explain the design of the consistent revision control system.

2.1 Model

The system consists of an a priori unknown number of clients and a storage server. The server provides an abstraction of consistent shared storage to the clients, who access it using operations to read and write data, and with operations to control different revisions of the data. We assume that all clients are correct and follow the protocol. The server may be faulty or *corrupted* and deviate from the protocol in arbitrary ways, but not break any cryptographic primitives.

The clients never communicate with each other directly, they communicate only via the server. This model is convenient and realistic because the clients are not required to know each other, the network topology may prevent direct communication between them, and they can operate independently of each other. Revision control systems enable a convenient form of computer-supported cooperative work, because the collaborators can contribute at different times and from different locations.

We assume that each client is identified by a public key/private key pair, signed by a trusted certification authority (CA). Every client trusts one or more CAs, whose root keys it stores in a local directory in the form of self-signed X.509 certificates. Clients identify each other only by their public key; more precisely, clients accept every public key as the identity of another client when the key is accompanied by a certificate from a trusted CA. The system distributes the keys among clients as needed; a client only needs the trusted CA keys before it starts to interact with the storage service. Representing client identities by keys simplifies key distribution considerably [13].

Every client maintains a small trusted memory, whose size is independent of the size of the shared storage space. In order to prevent a corrupted server from introducing unauthorized modifications to the shared data, clients sign all their write operations and verify the integrity of the data they read using digital signatures. But since the clients do not communicate with each other, we cannot prevent that the server completes a write operation of one client, and still returns stale data to another client.

The notion of *fork-linearizability* provides the next-best notion of consistency in this model [5,14]. It ensures that all operations in the view of every client are legal in the sense that data returned by a read operation has been written by the indicated client, and that when the server causes the views of two clients to differ, even in a single operation only, then these clients may not see any further operation of each other afterwards.

Our goal is to implement a storage service that provides read and write operations, which execute atomically and according to their specification whenever the server is correct; when the server is faulty, the storage service still provides fork-linearizability. We refer to the work of Cachin et al. [5] for a formal notion that captures this requirement under the name of a *fork-linearizable emulation* of a storage service on a potentially corrupted server. In the subsequent sections, we explain how fork-linearizable storage is implemented by our consistent storage service and by our consistent revision control service.

Naturally, a corrupted server may simply refuse to cooperate, and then the clients will have to reconstruct the shared data from their own records. But this attack cannot be prevented. There is no easy solution to this problem, except to choose a more trustworthy server.

On the other hand, a fork-linearizable emulation ensures that the server cannot violate the consistency of the storage service and hide this attack from clients that are suspicious. Even if the clients communicate out-of-band only occasionally, for example, by sending email to each other directly, or through a discussion forum on a project website, they are guaranteed to immediately discover any inconsistencies that were introduced ever by a faulty server.

A more subtle attack occurs when the server conspires with a client and violates the assumption that clients are correct. The current design does not prevent such behavior, but our system provides some means that help the correct clients to recover from such attacks. We discuss these at the end of the paper (Section 5).

2.2 Consistent Storage Service

The *consistent storage service* (CSS) provides a simple interface for reading and writing short byte arrays and ensures fork-linearizability with an untrusted server. There is no hard limit on the size of the stored byte arrays, but the service is designed for sizes up to 10 or 100 KiB because all values are transiently kept in main memory.

CSS provides one storage location for every client, called a *register*. The client is the only one who may write to its register, but all clients may read from it, and there is an operation that reads all registers in a single step. Formally, CSS combines an array of single-writer/multi-reader registers [10] with an atomic snapshot object [1].

The service provides the following interface to clients, expressed as method invocations:

getkeys() returns a list of all client identities that are known to the server so far, represented by their public keys. The server learns the identity of a client as soon as the client invokes its first method. A client may use the output of the operation in subsequent queries.

write(*data*) stores *data* in the register of the client at the server, overwriting data previously stored there. In CSS, a client may only write to its own register.

read(*key*) reads the register identified by the public key *key* and returns the stored data. If no such data exists, the operation returns none.

readall() reads all registers in one step and returns a list of pairs (*key*, *data*), representing all registers stored by the server; every pair contains the corresponding client *key* and the stored *data*. This method is equivalent to invoking getkeys(), followed by invoking read(*key*) for all *key* values returned, all in one atomic step. Its purpose is to give a consistent view of all registers.

We use the *lock-step protocol* of Cachin et al. [5] to implement CSS. The protocol is noteworthy for using the server only as intermediary storage; in particular, the server does not perform any cryptographic operations. The protocol

Preliminaries. CSS stores a register value $data_{key}$ for each client identified by *key*. Only the client identified by *key* may write to $data_{key}$, but every client may read from any register. Every client locally maintains a timestamp that it increments during every operation. We call an array of timestamps a *version*; a version is an associative array V that maps keys to timestamps, denoted by $V[key] = t$. We write $V[key] = \perp$ if $V[key]$ is not defined. Versions acts as a vector clock for ordering operations. Two versions V and W are ordered so that V is *smaller than or equal to* W whenever $V[key] \leq W[key]$ for all values *key* such that $V[key] \neq \perp$.

Client state. The client maintains a version T representing its last completed operation. Note that a client identified by *key* finds its own timestamp in $T[key]$.

For simplicity of the description, we assume the client also keeps a copy of its own data value $data_{key}$ and writes it back during every read operation. (In the implementation, it only stores a collision-resistant hash of the data value and sends that in a read operation; in a write operation, it sends the data value.)

Server state. The server stores the register values in an associative array X, where entry $X[key]$ contains $(data_{key}, \sigma_{key})$, representing the register value and a digital signature issued under *key* on the string $\mathsf{value} \,\|\, data_{key} \,\|\, t$, where t is a timestamp equal to $T[key]$ when the client completed the operation that wrote $data_{key}$.

The server also keeps information from the last completed operation: the version V associated to it, the key *last* identifying the client performing the operation, and a digital signature ω under key *last* on $\mathsf{commit} \,\|\, V$.

Operation. When a client identified by *key* invokes a write, read, or readall operation, it sends the request together with *key* in a *submit message* to the server. The server sends a *reply message*, containing the version V, the key *last*, and the accompanying signature ω from the last operation. In a read operation for register identified by *rkey*, the server also sends the register value $X[rkey] = (data_{rkey}, \sigma_{rkey})$. In a readall operation, the server adds all register values X. The server then waits for a *commit* message from this client and does not process any messages from other clients.

The client verifies that the reply message contains valid data: the version V must be at least as large as its own version T, the entry $V[key]$ must be equal to its own timestamp $T[key]$, and the signature ω on $\mathsf{commit} \,\|\, V$ must be valid under key *last*. In a read or readall operation, the client also verifies that σ_{rkey} is a valid signature under *rkey* on the string $\mathsf{value} \,\|\, data_{rkey} \,\|\, V[rkey]$, either for only one *rkey* in a single-register read or for all values *rkey* such that $X[rkey] \neq \perp$ in a readall operation. When the client detects any inconsistency in the reply, it considers the server to be faulty, generates an alarm, and aborts.

After the client has successfully verified the reply, it adopts the received version V as its own version T, increments its timestamp $T[key]$, and signs the new version T, resulting in a signature φ. It issues another signature σ on $\mathsf{value} \,\|\, data_{key} \,\|\, T[key]$, binding its data value to the timestamp. Then it sends a *commit message* to the server, containing T, φ, $data_{key}$, and σ.

When receiving the commit message, the server stores T, *key*, and φ as its version V, client key *last*, and signature ω that represent the last operation. The server also updates $X[key]$ with the received value $data_{key}$ and σ.

Fig. 1. The implementation of CSS using the lock-step protocol (adapted from [5,14]).

has been modified from using a fixed number of clients to handle an a priori unbounded number of clients that are identified only by a public key. Instead of using vectors, versions are represented by an associative array that maps every known client key to the corresponding timestamp. The clients maintain some state in their local memory and save it on persistent storage between operations. The protocol is shown in Figure 1.

The lock-step protocol has the drawback of not being wait-free [10] because when the server waits for the *commit* message from a client, no other client can proceed with an operation. Mazières and Shasha [14] and Cachin et al. [5] both present seemingly more efficient protocols that allow some client operations to proceed in parallel. However, it has been shown that in all fork-linearizable storage emulation protocols, a reader must wait for a concurrent writer [5][3].

We therefore chose to implement CSS with the lock-step protocol for the following reasons: First, the addition of the readall operation introduces the above conflict between readall and *every* write operation. We know that our consistent revision control application (described in Section 2.4) will use only write and readall operations, and we expect that they occur about equally often. Hence, the potential for exploiting concurrency is reduced to concurrent read operations. Second, the protocol allowing for concurrent operations is considerably more involved than the lock-step protocol. The small potential gain did not merit the added implementation complexity.

2.3 Revision Control

A *revision control system* (RCS) provides operations for storing and retrieving multiple versions of the same set of documents. It facilitates collaboration among multiple users, who may work independently with the information. The RCS assigns revision numbers to the documents and maintains a history of all versions. The documents usually consist of a hierarchical set of files and directories in a file system. Revision control systems are an important collaboration tool, as can be seen from the large number of existing systems (Wikipedia's "List of revision control software" lists 64 systems as of Sept. 2008).

For the purpose of designing our consistent RCS, we describe here the main features of a *generic* centralized RCS. A *centralized* RCS uses a dedicated server for controlling revisions and storing the history, in contrast to a *distributed* RCS, where this task is shared by all users. Our RCS is modeled after two popular RCS for source code, CVS (`http://www.nongnu.org/cvs/`) and Subversion (`http://subversion.tigris.org/`); they both allow users to update the same document concurrently.

We expect the client interface of an RCS to provide the following main operations:

Checkout: A checkout operation transfers all documents from the server repository to the client. It creates a copy of the files and directories on the client,

[3] Weaker semantics than fork-linearizability can give rise to wait-free storage emulation protocols [4].

called the *working copy*. All editing takes place there. The RCS also supports attributes attached to documents and version control for them.

Commit: After adding, modifying, or deleting some files in the working copy, the client wants to transfer the changes back to the central server, thereby making the changes visible to other clients. The client does this with a commit (or checkin) operation. Its effect is to create a new *revision* with a distinct identifier, called the *revision number*. We assume that revision numbers in a sequence of commits issued by multiple clients increase monotonically over time.

Update: An update operation transfers the most recent revision of all files from the server to the client and updates the working copy accordingly. The system also supports updating to a revision with a particular revision number.
When a client has modified some files locally and wants to commit the changes, it may have to perform an update, before the RCS allows a commit operation. This happens when some modifications of the client overlap and *conflict* with modifications committed by others. In this case, the commit operation will fail, the client is told to first update its working copy and to merge the concurrent changes, before the client may attempt another commit.

Typical RCS also support operations to populate the server repository with a set of documents initially, to rename repository contents, to create branches and merge them again, and to tag revisions with keywords. These operations may be present, but are not our main focus because they can be expressed as variations of the above three main operations.

We assume that all operations are transactional so that their changes either take effect in one atomic step on the server, or leave no trace in the repository in case of a failure.

2.4 Consistent Revision Control

Our *consistent revision control system* (CRCS) implements a revision control system that protects the integrity of the repository against a corrupted server. CRCS provides the same operations as an ordinary RCS and emulates a fork-linearizable storage service on the repository. We achieve fork-linearizability in terms of the checkout and update operations of CRCS, which implement a read operation on the repository, and in terms of the commit operation, which implements a write operation on the repository.

Fork-linearizability for a revision control system guarantees the following. Suppose a client A updates its working copy with CRCS to some revision number r. If A sees even a single file that was committed by another client B in revision r, then fork-linearizability implies that all files in client A's working copy have been cryptographically verified and are equal to those committed by B in revision r. Conversely, if there exists a more recent revision $s > r$ committed by a third client C, and the server hides revision s from A, then A can never again update to any revision committed by C or by anyone who updated

to s. Because of this all-or-nothing implication of fork-linearizability, one can very easily detect even subtle modifications of a single file by a corrupted server.

We implement CRCS by combining our CSS with an unmodified RCS. CRCS computes a hash tree [15] over the set of documents in the repository and basically stores the root hash of the tree using CSS. This construction extends the integrity guaranteed by CSS from the root hash to the entire data. Suppose every client commits changes to CRCS by first committing its working copy using RCS, thereby obtaining a revision number r, computing the new root hash h, and then writing the tuple (r, h) to its register. This stores all information in CSS that another client needs for updating its working copy to the most recent revision and for verifying its integrity. But because CRCS also supports cryptographically verified update operations for previous revisions in the repository, the design is more complex.

Every client maintains a *revision log* L with information about every revision that it committed. The revision log is a list of tuples (r, h, c), denoting the revision number r, the root hash h, and a *revision commitment* c, sorted chronologically (i.e., according to r). Let H denote a collision-free cryptographic hash function. The revision commitment binds together all previous commit operations of the client in a hash chain; when committing revision r with hash h, the client computes c as $H(r \,\|\, h \,\|\, c')$, using the revision commitment c' from the last tuple in L (or $c' = \perp$ if L is empty). The same chaining scheme has been used in many other timestamping and data authentication algorithms [9].

For the description of the CRCS algorithm below, assume that every client stores its complete revision log in L. For increased efficiency, an implementation may actually maintain only the last tuple of L in CSS and keep the rest of L in untrusted shared storage; the collision resistance of H guarantees the uniqueness of every revision log given its last revision commitment.

The client proceeds now as follows to implement the main operations of CRCS. If one of the checks in the algorithm fails, the client generates an alarm and aborts.

Checkout: To check out the highest revision, the client invokes the readall() operation of CSS and determines the largest revision number r from the returned revision logs and the corresponding root hash h. After invoking checkout of RCS for revision r, the checkout algorithm recomputes the hash tree on the working copy and verifies that its root hash is equal to h.

Commit: The client first calls the update operation of CRCS (see below) to bring its working copy to the most recent revision according to CSS. Then it commits the working copy with RCS to obtain a new revision number r. If this fails, the operation aborts and the client is told to update and to try again. If all goes well, the client computes the root hash h of the hash tree on its working copy, extends the client's revision log L with r and h, and invokes write(L) from CSS.

Update: The update operation is very similar to checkout. The client performs readall() to obtain all revision logs, determines the largest revision number r with corresponding root hash h, calls update from RCS to bring its working

copy to revision r, recomputes the changed paths in the hash tree, and verifies that the root hash matches h.
For updating to a particular revision r, the algorithm determines the client that committed r from all revision logs, locates the corresponding tuple $(r, h, \cdot)$ in some revision log L, and verifies L by following the hash chain from the tuple with r to the end of L. Then it proceeds as above, updating to revision r from RCS and verifying the working copy with respect to h.
When recomputing the hash tree for files that have changed in the repository, it is important that the client does that on a clean working copy, before the modifications from its working copy are applied. As the RCS merges the updates with the client's own changes, the update operation creates a working copy that differs from revision r in the repository.

Because the operations of CRCS verify that the working copy is consistent with the revision numbers and their root hashes maintained by CSS, the fork-linearizability of CSS implies the same property also for CRCS.

Note that the above algorithm introduces no new race conditions compared to RCS. As a consequence of synchronizing the client with CSS and RCS, it would be possible to create such problems. But the atomicity of the operations on CSS ensures that the more complex operations of CRCS are also atomic. In particular, whenever a client invokes checkout or update and retrieves some revision number from CSS, it always finds this revision in the repository of RCS. This holds because the commit operation of RCS precedes the writing of the corresponding revision number to CSS. Of course, there may already exist a more recent revision in the repository of RCS in the mean time, but this may also happen in the generic RCS, when another commit operation occurs immediately after an update.

3 Implementation

We have implemented our design in Python on Unix in two parts: first, the consistent storage service and, second, the consistent revision control system. The Python programming language encourages the kind of rapid prototyping we wanted and allowed a very natural transcription of the protocols. We chose *Subversion* (SVN) as the lower-level revision control system because it is widely used and because it fits our model of an RCS from Section 2.3. Hence, we refer to our implementation as *Consistent Subversion* (CSVN). Cryptographic operations are provided by OpenSSL via the M2Crypto Python interface to OpenSSL [18].

3.1 Consistent Storage Service

The implementation of CSS according to Section 2.2 stores arbitrary byte arrays. It is available as a library to clients. We wrote a simple interactive client application to read and write values entered by the user. The rich syntax of Python resulted in the server part of the algorithm in Figure 1 consisting of

about 250 lines of code and the client part consisting of about 200 lines of code, including the operations for key management. Having a succinct implementation is important for maintainability, and especially important for security-relevant software.

CSS uses Python's object serialization over TCP connections for transport. The server implementation is single-threaded according to the lock-step protocol; it uses a time-out in order to tolerate a client that crashes between sending a submit message and sending the corresponding commit message. We plan to integrate SSL/TLS support for increasing the security of the client-server connections in the future; currently, network attacks appear to the clients as server faults.

3.2 Consistent Revision Control with Subversion

We implemented CSVN in the form of a library that interfaces to SVN and provides the three main revision control operations. The operations invoke our consistent storage service and the Python SVN Extension (`http://pysvn.tigris.org/`). The SVN server remains unchanged. We also created small wrapper scripts for a user to invoke the client operations. The CSVN library consists of about 170 lines of code, and the scripts of about 75 lines of code each. Hence, the code is very compact.

For the description below, let a *path* denote the unit of information managed by SVN; a path may be a directory containing other paths, a file, or a symbolic link.

Hash Trees. The protocol requires to compute a hash tree over the documents in a revision. Let us define a hash function $\mathcal{H}$ on paths maintained by SVN. The hash value of a path p that represents a file or a symbolic link is defined as

$$\mathcal{H}(p) = H\big(H(p) \parallel H(C(p))\big),$$

where $C(p)$ is the content of p. The hash value of a path p representing a directory is

$$\mathcal{H}(p) = H\big(\mathcal{H}(p_1) \parallel \mathcal{H}(p_2) \parallel \ldots \parallel \mathcal{H}(p_n) \parallel H(p)\big),$$

where $p_1, \ldots, p_n$ is a sorted list of all paths in p. We denote the root hash of a repository by $\mathcal{H}(\text{“.”})$.

It would be prohibitively expensive to recompute the hash values of all paths in a large repository upon every change of a single file. Therefore, the client stores the hash value of every path as an SVN property of the path. During an update operation, CSVN recomputes the hash values of all changed files and of all directories along the path from the changed files to the root. For a repository with n files, this reduces the cost of updating m modified files from linear in n to $\mathcal{O}(m + d)$, where d is the maximum affected depth in the directory tree.

The hash values are stored on the SVN server because properties are revision-controlled in SVN. Note that storing them on untrusted storage is unproblematic. The hash values are not actually needed by a client who checks out the complete

repository because the client recomputes the entire hash tree anyway during verification. But they are needed for partial checkouts, as explained below.

Integration with SVN. During checkout and update operations, CSVN installs a callback before invoking the SVN library, which collects all relevant events reported by SVN; such relevant events are the addition, update, and deletion of a path. Then CSVN invokes CSS to obtain a revision number r and retrieves revision r from SVN, as described in Section 2.4. To recompute the root hash of the working copy, CSVN traverses the working copy, but visits only paths for which a relevant event was collected during the SVN operation.

For a commit operation, CSVN first determines the modified paths which are going to be written to the repository. It does that with an SVN "info" operation that outputs a collection of changed paths. Then it traverses the working copy, visiting and recomputing hash values only for changed paths, and getting hash values for unchanged paths from their SVN properties. This yields the root hash $h = \mathcal{H}(".")$. CSVN further invokes the "commit" operation of SVN to write the updates to the repository and to obtain the new revision number r. Finally, it retrieves the revision commitment c from the last tuple in L, appends $(r, h, H(r \parallel h \parallel c))$ to L, and writes L using CSS.

This completes the description of the main CSVN operations. Further SVN operations can be implemented easily using the CSVN library and the three main CSVN operations.

The description so far assumes that clients always check out and update the complete file set in the repository at once. But this is not required in SVN, where a client may check out only a subdirectory from a repository, or commit only a subset of its working copy. The revision number and the root hash stored in CSS are always global properties of the repository, though. Operations on the partial repository are supported by our design and rely on the hash values stored in the SVN properties. For example, to check out a subtree from a repository, CSVN also needs to read all files along the path from the subtree's root to the repository root before it can verify the root hash.

An important and nice feature of this implementation is that it does not add any additional SVN server operations; because they usually involve the network and contain a cryptographic authentication operation during login, they tend to be rather slow.

4 Evaluation

We report on benchmarks to measure the performance of CSVN client operations in comparison to an unmodified SVN client. Since every operation of CSVN also invokes the corresponding operation of SVN, we are primarily interested in the overhead of CSVN over SVN.

We report on two kinds of performance evaluations: an application benchmark using real-life file sets of different sizes and a synthetic benchmark with artificially made-up file sets. Each benchmark consists of a series of tests executed by two clients, called A and B, where each test uses different data. For

each test, we run the unmodified SVN client and the CSVN client 20 times in succession and measure the average time taken by each step in the test. Every run starts with an empty repository and a freshly initialized CSS. Each test uses a pair of related file sets; we are interested in the time it takes to update a working copy and the repository from one file set to the other one.

Each run in a test consists of the following steps:

1. Client A initializes a new empty repository on the server. This step is the same for both systems, so we do not measure it.
2. *Create* — client A checks out revision 0, creating a working copy from the empty repository.
3. *Import* — client A copies the first file set into its working copy, adds it to the repository, and commits the changes; we measure the time for the commit operation only.
4. *Checkout (CO) all* — client B checks out the content of the repository into its own working copy; the working copies of A and B are now identical.
5. Client B modifies its working copy to reflect the second file set. This involves adding the files contained only in the second file set, deleting the files only present in the first set, and copying the changed files from the second set into the working copy. This step is identical for both systems and is not measured.
6. *Commit (CI) diff* — client B commits the changes in its working copy.
7. *Update (UP) diff* — client A, whose working copy still contains the first file set, updates it to the most recent revision in the repository, which contains the second file set; the working copies of A and B are again identical.

This sequence of steps is designed to capture the overhead of committing and updating a large file set at once (in the *import* and *checkout all* steps) and of committing and updating smaller number of files in a larger file set (in the *commit diff* and *update diff* steps).

The benchmarks use two separate hosts, one for the server and one for both clients; they are connected by a gigabit LAN. The machine for the clients is an IBM x345 system with 2 GiB of RAM and two hyper-threaded Intel Xeon CPUs (3.06 GHz clock speed). The machine for the server is an IBM x335 system with 2 GiB of RAM and two hyper-threaded Intel Xeon CPUs (2.80 GHz clock speed). Both machines have a single IBM Ultra320 SCSI disk with 73.4 GB capacity and run Debian GNU/Linux 4.0 with kernel 2.6.18 and Subversion 1.5.2. The SVN server is accessed using SSH and all data is stored on the local filesystems. We use the SHA-1 hash function and 1024-bit RSA for signatures.

4.1 Application Benchmark

The file sets in our application benchmark are different versions of the Linux kernel source tree, as reported in Table 1. All files can be downloaded from the Linux kernel archive (`http://kernel.org/`). We choose them since they represent a realistic directory structure and because the repository sizes range

over several orders of magnitude, from 632 KiB to 62 MiB. We selected the four versions that make up the first file set in a test based on their relative size. For each test, we pick the subsequently released version of the Linux kernel and use it as the second file set. The results are shown in Table 2.

Table 1. The four tests of the application benchmark and the used Linux kernel version pairs. The third and fourth columns list the number of files in the first file set and the number of changed (added, modified, or deleted) files between the two file sets, respectively.

Test (version pair)	Size	Files	Changed
0.11 → 0.12	0.63 MiB	100	91
1.0 → 1.0.1	5.9 MiB	561	12
2.0.1 → 2.0.2	27 MiB	2021	28
2.2.0 → 2.2.1	62 MiB	4599	10

Table 2. Results of the application benchmark. The numbers denote average elapsed time and standard deviation in seconds for SVN and CSVN in 20 runs, and the ratio of the two average times.

Step	SVN	CSVN	Ratio
Create	1.18 ±0.09	1.53 ±0.33	1.30
Import	0.93 ±0.02	1.94 ±0.00	2.08
CO all	0.99 ±0.00	1.10 ±0.00	1.11
CI diff	1.50 ±0.02	2.30 ±0.29	1.53
UP diff	0.94 ±0.00	1.05 ±0.01	1.11

Test 0.11 → 0.12

Step	SVN	CSVN	Ratio
Create	1.05 ±0.05	1.45 ±0.38	1.38
Import	3.70 ±0.11	6.76 ±0.00	1.83
CO all	1.98 ±0.00	3.17 ±0.48	1.60
CI diff	1.24 ±0.42	2.03 ±0.01	1.63
UP diff	0.94 ±0.01	1.11 ±0.00	1.17

Test 1.0 → 1.0.1

Step	SVN	CSVN	Ratio
Create	1.46 ±0.17	1.34 ±0.25	0.92
Import	14.08 ±0.71	28.84 ±1.02	2.05
CO all	7.49 ±2.38	12.31 ±2.44	1.64
CI diff	3.89 ±1.59	5.15 ±0.47	1.32
UP diff	0.94 ±0.00	2.28 ±0.02	2.42

Test 2.0.1 → 2.0.2

Step	SVN	CSVN	Ratio
Create	0.68 ±0.06	1.18 ±0.29	1.72
Import	36.35 ±1.05	79.26 ±1.29	2.18
CO all	13.38 ±2.04	29.28 ±2.28	2.19
CI diff	10.20 ±3.68	9.64 ±2.53	0.95
UP diff	1.27 ±1.53	1.69 ±0.49	1.33

Test 2.2.0 → 2.2.1

4.2 Synthetic Benchmark

In this benchmark, we wish to measure how the running time changes when we grow the directory structure in a repository from one directory to a large tree,

Table 3. Results of the synthetic benchmark. The numbers denote average elapsed time and standard deviation in seconds for SVN and CSVN in 20 runs, and the ratio of the two average times.

Step	SVN	CSVN	Ratio
Create	1.48 ±0.12	1.35 ±0.40	0.91
Import	2.71 ±0.06	4.19 ±0.50	1.55
CO all	1.99 ±0.00	2.90 ±0.01	1.46
CI diff	1.87 ±0.22	3.02 ±0.01	1.61
UP diff	0.95 ±0.00	1.73 ±0.01	1.83

Depth 0

Step	SVN	CSVN	Ratio
Create	1.25 ±0.06	1.25 ±0.39	1.00
Import	2.01 ±0.33	3.89 ±0.01	1.93
CO all	1.99 ±0.00	2.40 ±0.01	1.21
CI diff	0.93 ±0.05	1.51 ±0.01	1.63
UP diff	0.95 ±0.00	1.01 ±0.01	1.07

Depth 2

Step	SVN	CSVN	Ratio
Create	1.27 ±0.01	1.46 ±0.39	1.15
Import	1.95 ±0.27	3.88 ±0.01	1.99
CO all	1.99 ±0.01	2.30 ±0.02	1.16
CI diff	0.92 ±0.06	1.61 ±0.03	1.75
UP diff	0.95 ±0.00	0.94 ±0.01	1.00

Depth 4

Step	SVN	CSVN	Ratio
Create	0.86 ±0.23	1.33 ±0.14	1.55
Import	4.33 ±0.42	10.59 ±0.89	2.44
CO all	8.75 ±1.52	9.95 ±1.14	1.14
CI diff	0.88 ±0.04	1.31 ±0.24	1.50
UP diff	2.28 ±1.15	1.90 ±0.51	0.84

Depth 8

but keep the number of files constant. To do this, we create four artificial file sets, each consisting of 256 files, each file of size 10 KiB, for a total data size of 2.5 MiB per file set. The files are filled with random pieces of C code taken from the Linux 2.2.1 kernel; this is to generate files looking like a real source tree. The files are stored in a directory structure of varying depth. We define a directory structure of depth d as a full binary tree of depth d and store $256/2^d$ files in each of the 2^d leaf directories.

Our file sets are four directory structures with depths 0 (all files in one directory), 2, 4, and 8 (every file in a separate directory). In each test, the second file set is identical to the first one, up to a random modification to one of the files in a leaf directory. The results are shown in Table 3.

4.3 Results

The results of both benchmarks show that CSVN adds an overhead of a factor that is generally less than 2 and usually also less than 1.5. In absolute terms, the *import* and the *checkout all* steps are the slowest operations because they involve all files. The *import* step generally incurs also the biggest overhead, usually around 2. But the overhead of the *checkout all* step is not noticeably different from the overhead of the remaining steps. Generally, CSVN adds only a moderate overhead to most operations compared to the normal SVN client.

Observe the bigger variation in the execution times of the tests with larger file sets. One reason for this effect may be that large data sets create more unexpected interactions with other programs due to swapping and disk operations

than small data sets that fit in the kernel's buffer cache. Such variations also explain the few overhead ratios smaller than 1.

Among the results of the application benchmark in Table 2, the second largest overhead (after the *import* step) usually occurs for the *commit diff* step. The overhead on the large file sets is not bigger than that on the smaller file sets. This clearly shows the benefit of using a hash tree when only a small part of a large file set is updated.

In the results of the synthetic benchmark in Table 3, observe the overhead of the *commit diff* and the *update diff* steps. In both steps, only a single file is changed. The CSVN client must then read the hash values of all sibling files to compute the new hash values for the directory. With the increasing depth of the directory structure, the number of sibling files drops from 255 to 0, and this is reflected in the decreasing overhead.

In summary, although a 50%–100% larger execution time for SVN operations is clearly noticeable by the clients, we believe it is a reasonable price to pay for the added guarantee of cryptographically verified data integrity. These results should serve as a lower bound for the efficiency of our design, because they were carried out with our straight-forward layered prototype implementation in Python. If the CSVN operations would be integrated with the SVN client library, the directory tree in the working would have to be traversed only once instead of twice; moreover, hashing could be integrated with the traversal and performed concurrently with receiving or sending data to the server. With such an integrated design, the cryptographic overhead is likely to vanish, as shown in other benchmarks of cryptographic storage and file systems [19].

5 Conclusions

Protecting data integrity against unauthorized modifications is an important aspect of networked storage systems. This paper presented a novel approach to securing the integrity of data stored in revision control systems, and demonstrated its feasibility with our Consistent Subversion (CSVN) prototype. Our evaluation shows that the overhead is reasonable.

The biggest threat to our system are client failures. Protecting the system from malicious clients is also the area where future work is needed.

Our implementation already tolerates client crashes; one or more malicious clients alone cannot harm the integrity *if* the service provider is correct — measures to prevent such behavior can easily be added [11], but have not been described in this work. A corrupted client conspiring with a corrupted service provider, however, may undermine fork-linearizability.

A first barrier against such an attack is the CA that must authorize all clients before they access the service. It is therefore a good idea to make the CA is a separate entity from the storage service. If the threat of such a client-server conspiracy attack becomes too serious, one might adopt the complex cross-checking of versions signed by different clients introduced in SUNDR [14]. Unfortunately, the SUNDR protocol involves a much higher communication overhead in every

operation. One should also develop an additional tool that helps the clients to recover from a server failure; it should automatically reconcile the state of the repository from the information held by the clients in their working copies and their local memories.

Acknowledgments

We are grateful to Idit Keidar, Alexander Shraer, and Marko Vukolić for many discussions and valuable comments.

This work was supported in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.

[2] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale. A fresh look at the reliability of long-term digital storage. In *Proc. 1st European Conference on Computer Systems (EuroSys)*, pages 221–234, 2006.

[3] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12:225–244, 1994.

[4] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, 2009.

[5] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 129–138, Aug. 2007.

[6] D. Clarke, G. E. Suh, B. Gassend, A. Sudan, M. van Dijk, and S. Devadas. Towards constant bandwidth overhead integrity checking of untrusted data. In *Proc. 26th IEEE Symposium on Security & Privacy*, 2005.

[7] CNET News. Red Hat, Fedora servers compromised. `http://news.cnet.com/8301-1009_3-10023565-83.html`, Aug. 2008.

[8] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2003.

[9] S. Haber and W. S. Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3:99–111, 1991.

[10] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.

[11] J. Li, M. Krohn, D. Mazires, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. 6th Symp. Operating Systems Design and Implementation (OSDI)*, pages 121–136, 2004.

[12] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proc. 4th Symp. Operating Systems Design and Implementation (OSDI)*, 2000.

[13] D. Mazières, M. Kaminsky, F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. 17th ACM Symposium on Operating System Principles (SOSP)*, 1999.

[14] D. Mazières and D. Shasha. Building secure file systems out of Byzantine storage. In *Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC)*, 2002.

[15] R. C. Merkle. Protocols for public-key cryptosystems. In *Proc. IEEE Symposium on Security & Privacy*, pages 122–133, 1980.

[16] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. *ACM Transactions on Storage*, 2(2):107–138, May 2006.

[17] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *Proc. 15th ACM Conference on Computer and Communications Security*, 2008.

[18] N. P. Siong and H. Toivonen. M2Crypto Python interface to OpenSSL. `http://chandlerproject.org/Projects/MeTooCrypto`, 2008. Version 0.18.2.

[19] C. P. Wright, J. Dave, and E. Zadok. Cryptographic file systems performance: What you don't know can hurt you. In *Proc. 2nd International IEEE Security in Storage Workshop (SISW)*, 2003.