Auditable Version Control Systems

Bo Chen Reza Curtmola
Department of Computer Science
New Jersey Institute of Technology
bc47@njit.edu crix@njit.edu

***Abstract*—Version control provides the ability to track and control changes made to the data over time. Software development often relies on a Version Control System (VCS) to automate the management of source code, documentation and configuration files. The VCS system stores all the changes to the data into a repository, such that any version of the data can be retrieved at any time in the future. Due to their potentially massive size, VCS repositories are often hosted at third parties which, unfortunately, are not necessarily trusted. Remote Data Checking (RDC) can be used to address concerns about the untrusted nature the VCS server by allowing a data owner to periodically and efficiently check that the server continues to store her data.**

To reduce the storage overhead, modern version control systems usually adopt "delta encoding", in which only the differences (between versions) are recorded. As a particular type of delta encoding, skip delta encoding can optimize the combined cost of storage and retrieval.

In this work, we introduce *Auditable Version Control Systems* (AVCS), which are VCS systems designed to function under an adversarial setting. We present the definition of AVCS and then propose RDC–AVCS, an AVCS scheme for skip delta-based VCS systems, which relies on RDC mechanisms to ensure all the versions of a file are retrievable from the untrusted VCS server over time. In RDC–AVCS, the cost of checking the integrity of all the versions of a file is the same as checking the integrity of one file version and the client is only required to maintain the same amount of client storage like a regular (non-secure) VCS system. We make the important observation that the only meaningful operation for real-world VCS systems which use delta encoding is *append* and leverage this observation to build RDC–AVCS. Unlike previous solutions which rely on dynamic RDC and are interesting from a theoretical point of view, we take a pragmatic approach and provide a solution for real-world VCS systems.

We build a prototype for RDC–AVCS on top of a popular open-source version control system, Apache Subversion (SVN), and implement the most common VCS operations. Our security analysis and experimental evaluation show that RDC–AVCS successfully achieves the desired security guarantees at the cost of a modest decrease in performance compared to a regular (non-secure) SVN system.

I. Introduction

Version control (also known as *revision control*) is the management of changes to collections of information, such as documents, computer programs, web pages, or configuration files. Version control provides the ability to track and control the changes made to the data over time. This includes the ability to recover an old version of a document. Software development often relies on a *Version Control System* (VCS) to automate the management of source code, documentation and configuration files. A VCS provides several useful features to software developers, such as: retrieve previous versions of the source code in order to locate and fix bugs, roll back to earlier versions in case the working version becomes corrupted, or allow team development in which multiple developers can work simultaneously on updates. In fact, a VCS is indispensable for managing large software projects. Popular version control systems include CVS [7], Subversion [4], Git [12], and Mercurial [15].

A version control system automates the process of version control. A VCS records all changes to the data into a data store called *repository*, so that any version of the data can be retrieved at any time in the future. Oftentimes, repositories are hosted by a third party, since they are potentially massive in size and cannot be stored and managed locally. For example, both Sourceforge [17] and Google Code [14] host repositories (based on Subversion or Git) for open-source projects, and GitHub [13] provides a paid service for Git repositories. Unfortunately, a third party is not necessarily trusted, for several reasons. First of all, the service providers may rely on a public cloud storage platform, rather than an internal infrastructure, to host their users' data. For example, file hosting service providers like Dropbox [8], Bitcasa [5], that offer version control functionality to the stored data, use Amazon S3 [1] as a back-end storage service. Secondly, the service providers are vulnerable to various outside or even inside attacks. Thirdly, the service providers usually rely on complex distributed systems, which are vulnerable to various failures caused by hardware, software, or even administrative faults [45]. Additionally, unexpected accidental events may lead to the failure of services, *e.g.*, power outage [18], [19]. In Sec. III-B, we provide additional arguments to support this threat model and the need to audit VCS systems.

Remote Data Checking (RDC) [26], [25], [43] can be used to address these concerns about the untrusted nature of a third party that hosts the VCS repository. RDC is a mechanism that has been recently proposed to check the integrity of data stored at untrusted third party providers of storage services. Briefly, RDC allows a client who initially stores a file with a storage provider to later check if the storage provider continues to store the original file in its entirety. This check can be done periodically, depending on the client's needs.

From the data owner's point of view, it should be possible to retrieve any previous version of the data, even if the repository is hosted at an untrusted VCS server. In a straightforward

NDSS '14, 23-26 February 2014, San Diego, CA, USA
 ISBN 1-891562-35-5
http://dx.doi.org/10.14722/ndss.2014.23184

	DPDP[40]	DR − DPDP[41]	RDC–AVCS
Communication (Commit phase)	$O(n + log(t))$	$O(n + 1)$	$O(n + 1)$
Server computation (Commit phase)	$O(n + log(t))$	$O(n)$	$O(nlog(t))$
Client computation (Commit phase)	$O(n + log(t))$	$O(n + 1)$	$O(n + 1)$
Communication (Challenge phase)	$O(logn + log(t))$	$O(1 + logn)$	$O(1)$
Computation (server + client) (Challenge phase)	$O(logn + log(t))$	$O(1 + logn)$	$O(1)$
Communication (Retrieve phase)	$O(n + log(t))$	$O(n + 1)$	$O(n + 1)$
Server computation (Retrieve phase)	$O(tn + log(t))$	$O(tn + 1)$	$O(nlog(t) + 1)$
Client computation (Retrieve phase)	$O(n + log(t))$	$O(n)$	$O(n)$
Client storage	$O(n)$	$O(n)$	$O(n)$
Server storage	$O(nt)$	$O(nt)$	$O(nt)$

TABLE I: Comparison of different RDC schemes for version control systems. t is the number of versions in the repository and n is the number of blocks in a version. The costs for the Commit and Retrieve phases are for committing and retrieving one version. The costs for the Challenge phase are for checking the integrity of all versions in the repository. DPDP and DR − DPDP are built on top of delta-based version control systems, whereas our RDC–AVCS scheme is built on top of skip delta-based version control systems.

application of RDC, if a file F has t versions, $\mathtt{F}_0$ through $\mathtt{F}_{t-1}$, then each file version can be seen as an independent file and the client can use RDC independently to check the integrity of each file version. However, this solution has prohibitive costs for several reasons. VCS repositories may store many versions and storage overhead would be very large if every version is stored in its entirety (*e.g.*, the source code for the `gcc` compiler [11] has over 200,000 versions). Moreover, the RDC costs associated with creating metadata and checking each version independently would be too large.

To reduce the storage overhead, modern version control systems adopt "delta encoding" to store versions in a repository: Only the first version of a file is stored in its entirety, and each subsequent version of the file is stored as the difference from the immediate previous version. These differences are recorded in discrete files called "deltas". Thus, if there are t versions of a file, the VCS server stores them as the initial file and $t - 1$ deltas. A popular version control system that uses a variant of delta encoding is Git [12]. Delta encoding optimizes the storage required to represent all the versions of a file. However, a delta encoded repository is not optimized towards retrieving individual versions: To retrieve version t, the VCS server starts from the initial version and applies all subsequent deltas up to version t, thus incurring a cost linear in t. Considering that source code repositories may have hundreds of thousands of versions (*e.g.*, `GCC` [11]), retrieving an arbitrary version can be burdensome on the server.

Skip delta encoding is a type of delta encoding which is further optimized towards reducing the cost of retrieval. A new file version is still stored as the difference from a previous file version; however, this difference is not relative to the immediate previous version, but it is relative to another previous version (more details in Sec. II-A). This ensures that retrieval of the t-th version only requires $log(t)$ applications of deltas by the VCS server. A popular VCS that uses skip delta encoding is Apache Subversion (in short, SVN) [4].

The evolution of a file managed with a VCS can be seen as a sequence of updates, each update resulting in a new file version. As such, the integrity of a VCS repository could be verified using an RDC protocol designed to allow dynamic updates to the data. Several RDC schemes can handle the full range of dynamic update operations [40], [49], such as modifications, insertions, and deletions. A dynamic RDC scheme can directly be used to check the integrity of the latest file version (every new file version can be seen as a series of updates to the previous file version). A dynamic RDC scheme can also be adapted to check the integrity of the entire VCS repository – basically check all versions of a file – by organizing the file versions in an authentication structure.

We argue that using a dynamic RDC scheme to check the integrity of a VCS repository has several important drawbacks:

– *First*, we make the observation that all real-world VCS systems *require only the append operation* – the repository stores the initial file version and a series of deltas for subsequent versions, all of which can be seen as append operations to the initial version. Thus, using a full-fledged dynamic RDC scheme that supports the full range of updates is overkill and incurs additional unnecessary overhead during the Challenge and Commit phases as illustrated in Table I. Indeed, previous work on checking integrity of version control systems [40], [41], [51] extends a dynamic RDC scheme which relies on a tree-like structure, thus adding a logarithmic cost to the Challenge and Commit phases. However, the only meaningful operation for modern VCS systems (*e.g.*, CVS, SVN, Git) is the append operation, since they are designed to keep a record of all the data in all previous versions.

– *Second*, a dynamic RDC scheme that supports the full range of dynamic updates has a higher complexity than an RDC scheme designed to only support appends at the end of the file. The additional complexity brings with it a more complex adversarial model and a more complex proof of security, all of which make the scheme more prone to security and implementation flaws.

Contributions. In this work, we propose RDC–AVCS, an auditable version control system designed to function even when the VCS repository is hosted at an untrusted party. Unlike previous solutions which rely on dynamic RDC and are interesting from a theoretical point of view, ours is the first to take a pragmatic approach for auditing real-world VCS systems. Our solution considers the format of modern VCS repositories, which leads to additional optimizations. Specifically, we make the following contributions:

- We give a technical overview of delta-based and skip delta-based VCS systems, which have been designed to work under a benign setting. We make the important

observation that the only meaningful operation in a real-world modern VCS system is *append*.

- We introduce the definition of *Auditable Version Control Systems* (AVCS), which are delta-based VCS systems designed to function under an adversarial setting. We then propose RDC–AVCS, an AVCS scheme for skip delta-based VCS systems, which relies on RDC mechanisms to ensure all the versions of a file are retrievable from the untrusted VCS server over time. Compared with previous solutions based on dynamic RDC, RDC–AVCS has several advantages. It is able to keep constant the cost of checking the integrity of all the versions in the VCS repository. This optimization is possible based on the important observation that the only meaningful operation in modern real-world VCS systems is append and based on the fact that RDC schemes designed for static data can securely support the append operation. RDC–AVCS is also conceptually much simpler, which simplifies the security analysis and reduces the possibility of implementation bugs. RDC–AVCS has the following features:
  - In addition to the regular functionality of a non-secure VCS system, RDC–AVCS offers the data owner the ability to check the integrity of all versions in the VCS repository.
  - The cost of checking the integrity of all the versions of a file is the same (asymptotically) with the cost of checking the integrity of one file version (*i.e.*, $O(1)$).
  - The data owner can check the correctness of a version retrieved from the VCS repository.
  - RDC–AVCS only requires the same amount of storage on the client like a regular (non-secure) VCS system.
- We build a prototype for RDC–AVCS on top of the popular open-source VCS system Apache Subversion (SVN). Our prototype, SSVN, implements the most common SVN operations. We also build a tool which facilitates the migration of non-secure SVN repositories to SSVN. Our experimental evaluation based on three representative SVN repositories (FileZilla, Wireshark, GCC) shows that SSVN incurs only a modest decrease in performance compared to a regular (non-secure) SVN system.

II. Background on Version Control Systems and Remote Data Checking

A. Version Control Systems

Software development relies on a *Version Control System (VCS)* to automate the management of source code, documentation and configuration files. Typically, one (or more) VCS clients interact with a VCS server and the VCS server stores all the changes to the data into a *main repository*, such that any prior version of the data can be retrieved at any time in the future. Each VCS client has a local repository, which stores the *working copy*, the changes made by the client to the working copy, and some metadata. The working copy is the version of the data that was last checked out by the client from the main VCS repository.

A VCS provides several useful features to track and control the revisions (changes) made to the data over time. This includes operations such as commit, update, revert, branch, merge, and log. In practice, the most commonly used operations by a VCS client are *commit* and *retrieve*. Commit refers to the process of submitting the latest changes of the data to the main repository, so that the changes to the working copy become permanent. Retrieve refers to the process of replacing the working copy with an older or a newer version stored on the server.

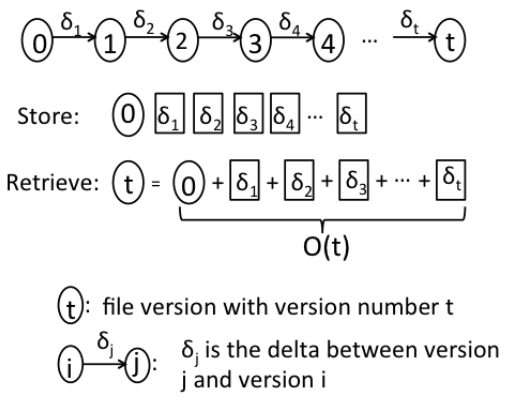
Delta-based VCS. With a version control system, the data owner would like to keep every change of her data in the repository, so that at any point of time in the future, she can revert to a previous version, or update to a new version. One simple solution is to store a new version of the data in its entirety upon each commit (*e.g.*, CVS [7] adopts this method for binary files). Such a straightforward solution, however, has large communication and storage overhead, since in most cases, only a small portion of the whole data has been updated; thus, sending and storing the whole new version may result in significant unnecessary communication and storage.

To reduce the storage overhead, modern VCS systems adopt "*delta encoding*" to store changes to the data in the repository: Only the first version of a file is stored in its entirety, and each subsequent version of the file is sent and stored as the difference from the immediate previous version. These differences are recorded in discrete files called "deltas". Thus, if there are t versions of a file, the VCS server stores them as the initial file and $t-1$ deltas (see Fig. 1(a)). Popular version control systems that use variants of delta encoding are Git [12], SVN [4] and CVS [7][1]. Delta encoding optimizes the storage required to represent all the versions of a file. However, a delta encoded repository is not optimized towards retrieving individual versions: To retrieve version t, the VCS server starts from the initial version and applies all subsequent deltas up to version t, thus incurring a cost linear in t (again, see Fig. 1(a)). Considering that source code repositories may have hundreds of thousands of versions (*e.g.*, `GCC` [11]), retrieving an arbitrary version can be burdensome on the server.

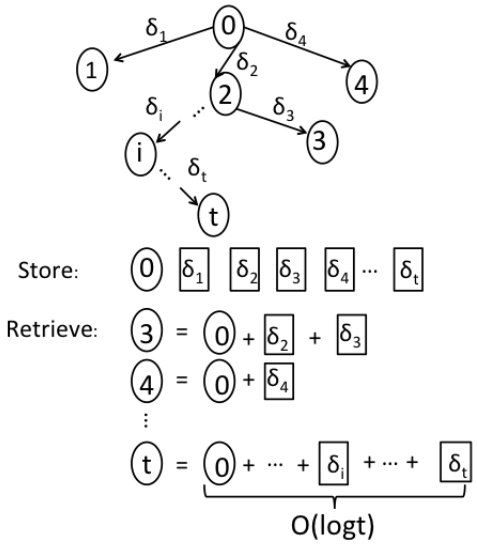
Skip delta-based VCS. Skip delta encoding is a type of delta encoding which is further optimized towards reducing the cost of retrieval. A new file version is still stored as the difference from a previous file version; however, this difference is not relative to the immediate previous version, but it is relative to a certain previous version. This ensures that retrieval of the t-th version only requires $log(t)$ applications of deltas by the VCS server. A popular VCS that uses skip delta encoding is Apache Subversion (in short, SVN) [4].

In this case, the difference is called a "skip delta" and the old version against which a new version is encoded is called a "skip version". When version i is committed, the skip delta is computed against the skip version j. The rule for selecting the skip version j is: Consider the binary representation of i and change the rightmost bit that has value "1" into a bit with

[1]CVS uses delta encoding only for text files



(a) Delta-based VCS



(b) Skip delta-based VCS

Fig. 1: Delta-based and skip delta-based version control systems.

value "0". For example, in Fig. 1(b), version 4's skip version is version 0, because the binary representation of 4 is 100, and by changing the rightmost "1" bit into a "0" bit, we get 0.

By adopting the skip delta-based approach, the cost to recover any version is logarithmic in the total number of versions. For example, in Fig. 1(b), to reconstruct version 3, start from version 0 and apply δ_2 and δ_3; to reconstruct version 4, start from version 0 and apply δ_4. The skip version for version 25 is 24, whose skip version is 16, whose skip version is 0. Thus, to reconstruct version 25, start from version 0 and apply $\delta_{16}, \delta_{24}, \delta_{25}$. In Appendix A, we show that the cost for retrieving an arbitrary version t is bounded by $O(log(t))$.

B. Remote Data Checking

Remote Data Checking (RDC) allows the data owner to check the integrity of data outsourced at an untrusted server, and thus to audit whether the server fulfills its contractual obligations. A remote data checking protocol consists of three phases: Setup, Challenge, and Retrieve. Consider that the storage of one file is outsourced at an untrusted server. Then, during the Setup phase, the data owner preprocesses the file $\mathtt{F}$ and generates verification metadata Σ, and then stores both $\mathtt{F}$ and Σ at the untrusted server. The data owner then deletes $\mathtt{F}$ and Σ from its local storage and only keeps a small constant amount of secret key material K. During the Challenge phase, a verifier (the data owner or a third-party verifier) challenges the server to prove that it really possesses the file previously stored by the data owner. The server generates a proof of possession based on the stored file and metadata, and sends back the proof. The client then checks the proof based on the key material K. During Retrieve, the data owner recovers the original file.

PDP (Provable Data Possession [26]) and PoR (Proofs of Retrievability [43], [46]) are two examples of RDC protocols. In PDP/PoR, during the Setup phase, the data is seen as a collection of fixed-size blocks, and the client computes a tag for each block. During the Challenge phase, the verifier randomly checks the integrity of a random subset of the file blocks. The Challenge phase can be very efficient: For example, it is shown [26] that if the server corrupts a certain fraction of the file (*e.g.*, 1%), the verifier can detect such corruptions with high probability by only randomly checking a constant number of blocks; in this case, the communication between the verifier and the server is also constant in size.

PDP/PoR have been shown to be extremely efficient during the Challenge phase [26], [43], [46], with constant communication and constant client/server computation. However, both PDP and PoR have been originally proposed for archival storage and only support static data. Later, a more complex PDP protocol was proposed to support dynamic operations on the outsourced data, such as insertions, deletions and modifications [40]. In Sec. V-A we show that RDC schemes for static data can securely support one specific dynamic operation, namely append at the end of the file. In section IV, we build an RDC scheme for skip delta-based VCS systems, which relies on any RDC scheme that supports block appends at the end of the file.

III. Model and Guarantees

A. System Model

An Auditable Version Control System (AVCS) is a version control system (VCS) designed to function under an adversarial setting. In AVCS, just like in a regular VCS, one or more clients store data at a server. The server maintains the *main repository*, where all the versions of the data are stored. Each client runs an AVCS client software. In this paper, we use the term client to refer to the AVCS client software and server to refer to the AVCS server software. Each AVCS client has a local repository, which stores the working copy, the changes made by the client to the working copy, and some metadata. The working copy is the version of the data that was last checked out by the client from the main VCS repository.

From a client's point of view, the interface exposed by the server includes two main operations: *commit* and *retrieve*[2].

[2]VCS systems permit additional operations such as *branch*, *merge*, *log*, etc., but in this paper we focus on *commit* and *retrieve*, which are the most common operations.

Commit refers to the process of submitting the latest changes of the data to the main repository, so that the changes in the client's working copy become permanent. Retrieve refers to the process of replacing the client's working copy with an older or a newer version stored on the server.

AVCS incorporates all the functionality offered by a regular VCS. In addition, the AVCS server exposes one additional operation, *check*, which permits the client to check if the server possesses all the versions of a file.

The AVCS main repository may contain several projects. Each project may contain one or more files. For each file, the changes submitted by the client are stored by the server using delta encoding, as described in Sec. II-A. Each change is stored as a discrete "delta" file. So, if there are $t-1$ changes for a file, then the server will store the initial version of the file and $t-1$ delta files, $\delta_1, ..., \delta_{t-1}$. We focus our discussion on storing, checking, and retrieving the versions of one file; this can be easily generalized to multiple files.

B. Adversarial Model

We consider a threat model in which there are no malicious clients, *i.e.*, all clients are trusted. However, the server is not trusted and may misbehave [26]. This captures a setting in which the employees of a company collaborate on a software development project (so they are all trusted), but the AVCS server is outsourced at a third party which is not necessarily trusted. The server may misbehave as follows:

– It may reclaim storage by discarding data that is rarely accessed (economically motivated), or try to hide data loss incidents to preserve its reputation. Data loss incidents may be accidental (e.g., administrative errors, hardware and software failures) or malicious (*e.g.*, insider or outsider attacks).

– During retrieve, it may not provide the requested version correctly, *e.g.*, it may provide a corrupted version, or a version which is either older or newer than the requested version. Possible reasons for such misbehavior could be: The repository has been corrupted (accidentally or maliciously), or the server has reclaimed some rarely accessed data, or the server-side software does not function properly, etc.

We consider a server that is rational and economically motivated. In this context, cheating is meaningful only if it cannot be detected and if it achieves some economic benefit (e.g., using less storage than required by the contract). We note that such an adversarial model is reasonable and captures many practical settings in which malicious servers will not cheat and risk their reputation, unless they can achieve a clear financial gain. In particular, we do not consider attacks in which the server simply corrupts a small portion of the repository (*e.g.*, 1 byte), because saving such a small amount of storage will not provide a significant benefit for the server. For a discussion about protection against small corruption attacks, see Sec. V.

The server is assumed to at least respond to the client's requests. Otherwise, if the server is non-responsive, the client will terminate its contract with the server and choose another service provider. To protect the client-server communication against external adversaries, we assume that this communication occurs over secure channels, *e.g.*, the communication is secured using SSL/TLS.

On the importance of auditing VCS systems. We provide several arguments to motivate this threat model and to highlight the importance of auditing VCS systems:

- Even though source code repositories are not very large (*e.g.*, the entire `gcc` repository is about 1GB), popular hosting services have a huge number of repositories. In 2013, GitHub hosted over 6 million repositories [6], SourceForge over 324,000 projects [22] and Google Code over 250,000 projects. It is conceivable that some service providers may be economically motivated to misbehave.
- The techniques we propose are applicable to all VCS-es that rely on skip delta encoding, including those that store other type of data than source code. For example, Dropbox saves the history of all deleted and earlier versions of files (free for 30 days, and unlimited deletion recovery and version history with the "Packrat" option).
- There are ongoing efforts to add support for large media binary files into VCS-es like Git [20], [21].
- Hosting providers like Dropbox [8] and Bitcasa [5] that offer version control functionality rely on cloud storage services like Amazon S3 as the back-end storage. It is conceivable that even providers like GitHub may adopt a similar model in the future. There is plenty of evidence that cloud service providers should not be fully trusted.

C. Security Guarantees

Consider an AVCS repository which contains t versions of the file $\mathtt{F}$ (these are stored in the repository as the initial version of the file $\mathtt{F}_0$ and $t-1$ delta files, δ_1, ..., δ_{t-1}). Let $\tilde{\mathtt{F}}$ be the virtual file obtained by concatenating $\mathtt{F}_0$, δ_1, ..., δ_{t-1}, *i.e.* $\tilde{\mathtt{F}} = \mathtt{F}_0||\delta_1||\delta_2||...\delta_{t-1}$. We seek to build AVCS systems which provide the following security guarantees:

– **SG1** (Data Possession): Upon checking the integrity of all the versions of $\mathtt{F}$ stored in the repository, the client can detect if the server corrupts a fraction of $\tilde{\mathtt{F}}$.

– **SG2** (Version Correctness): Upon retrieving $\mathtt{F}_i$ (version i of $\mathtt{F}$) from the server, the client can verify the correctness of $\mathtt{F}_i$, for any $i \in [0, t-1]$.

The practical implications of these guarantees are that the server cannot corrupt some of the file's versions without being detected and that it cannot serve an incorrect file version to the client. **SG1** captures the client's ability to check if the server continues to possess all of the versions of $\mathtt{F}$ that have been stored in the main repository. **SG2** captures the client's ability to detect if the server provides a corrupt version, or a version that is different than the version requested by the client.

IV. Auditable Version Control Systems (AVCS)

In this section, we first give an overview of VCS systems designed to work under a benign setting. We then introduce the definition of *Auditable Version Control Systems* (AVCS), which are VCS systems designed to function under an adversarial

setting, and propose a construction based on remote data checking mechanisms.

Notation. The VCS repository contains t versions of the file $\mathtt{F}$, which are stored in the repository as $\mathtt{F}_0, \delta_1, \delta_2, ..., \delta_{t-1}$. $\mathtt{F}_0$ is the initial version of the file, and the $t-1$ delta files are based on skip delta encoding as described in Sec. II-A. We focus our discussion on storing, checking, and retrieving the versions of one file; this can be generalized to multiple files.

We use $\mathtt{F}_i$ to denote version i of the file. We use $\mathtt{F}_{skip(t)}$ to denote the skip version for $\mathtt{F}_t$ (the algorithm for determining $\mathtt{F}_{skip(t)}$ is described in Sec. II-A). We write $\mathtt{F}_i = \mathtt{F}_j + \delta$ to denote that $\mathtt{F}_i$ is obtained by applying δ to $\mathtt{F}_j$.

A. Skip Delta-based Version Control Systems

Version control systems which use skip delta encoding have been designed for a benign setting, in which the VCS server is assumed to be fully trusted. A popular VCS which relies on skip delta encoding is Apache Subversion [4] (in short, SVN), described on its website as an "open-source, centralized version control system characterized by its reliability as a safe haven for valuable data".

The main operations of such VCS systems fall under three phases: Setup, Commit, and Retrieve, as follows:

In the Setup phase, the client (data owner) contacts the server to create a new project in the main VCS repository[3]. For example, in SVN, this can be achieved using the command "`svn import`", which will create a new project in the main VCS repository using a codebase that exists at the client – this will be the first version of the project. The client will then create its local working copy by checking out this first version from the server, using the command "`svn checkout`".

In the Commit phase, the client commits the changes in its local working copy into the main VCS repository. For example, in SVN, this can be achieved using the command "`svn commit`". The client wants to commit a new version, $\mathtt{F}_t$ (note that the client also has a local copy of $\mathtt{F}_{t-1}$, which is the working copy). Then the client computes the "delta" between $\mathtt{F}_t$ and $\mathtt{F}_{t-1}$, *i.e.* δ such that $\mathtt{F}_t = \mathtt{F}_{t-1} + \delta$, and sends δ to the server. After receiving δ, the server executes:

1) Compute $\mathtt{F}_{t-1}$ based on data in the repository (*i.e.*, start from $\mathtt{F}_0$ and apply skip deltas $\ldots, \delta_i, \ldots, \delta_{t-1}$).
2) Compute $\mathtt{F}_t$ based on $\mathtt{F}_{t-1}$ and δ: $\mathtt{F}_t = \mathtt{F}_{t-1} + \delta$.
3) Compute the skip version $\mathtt{F}_{skip}$ based on the data in the repository (*i.e.*, start from $\mathtt{F}_0$ and apply skip deltas $\ldots, \delta_i, \ldots, \delta_{skip}$).
4) Compute δ_{skip} such that $\mathtt{F}_t = \mathtt{F}_{skip} + \delta_{skip}$, and store δ_{skip} as δ_t in the repository.

In the Retrieve phase, the client retrieves an arbitrary version of the data. For example, in SVN, this can be achieved using the "`svn update -r` i" command. The client wants to replace version j (the working copy) with version i. The server executes:

1) Compute $\mathtt{F}_i$ based on the data in the repository (*i.e.*, start from $\mathtt{F}_0$ and apply the corresponding skip deltas).
2) Compute $\mathtt{F}_j$ based on the data in the repository (*i.e.*, start from $\mathtt{F}_0$ and apply the corresponding skip deltas).
3) Compute δ such that $\mathtt{F}_i = \mathtt{F}_j + \delta$.
4) Return δ to the client.

The client then computes $\mathtt{F}_i$: $\mathtt{F}_i = \mathtt{F}_j + \delta$.

B. Definition of an AVCS system

The previous section described the behavior of a skip delta-based VCS system in a benign setting, where the VCS server is fully trusted and does not deviate from the protocol. However, as described in the adversarial model (Sec. III), in this work we consider a setting in which the VCS server is untrusted. We propose an *Auditable Version Control System* (AVCS), which is a delta-based VCS enhanced to work in an adversarial setting.

An AVCS scheme consists of seven polynomial-time algorithms (KeyGen, ComputeDelta, GenMetadata, GenProof, CheckProof, GenRetrieveVersionAndProof, CheckRetrieveProof). KeyGen is a key generation algorithm run by the client to setup the scheme. ComputeDelta is run by the client to compute a delta when committing a new file version. GenMetadata is run by the client to generate the verification metadata for a new file version, before committing the new version. GenProof is run by the server and CheckProof is run by the client in order to generate and verify a proof of data possession, respectively. Similarly, GenRetrieveVersionAndProof is run by the server and CheckRetrieveProof is run by the client to retrieve an arbitrary file version.

An AVCS system has four phases: Setup, Commit, Challenge, and Retrieve.

– Setup: The client runs KeyGen to generate the private key material and performs other initialization operations.

– Commit: To commit a new file version, the client runs ComputeDelta and GenMetadata to compute the delta and the metadata for the new file version, respectively. The delta and the metadata are both sent to the server.

– Challenge: Periodically, the verifier (client) challenges the server to obtain a proof that the server continues to store all the file versions committed by the client. The server uses GenProof to compute a proof of data possession, and the client uses CheckProof to validate the proof.

– Retrieve: The client requests an arbitrary version of the stored data. The server runs GenRetrieveVersionAndProof to obtain the requested file version, together with a proof of correctness. The client verifies the correctness of the file retrieved from the server by running CheckRetrieveProof.

Note that this definition encompasses VCS systems that use delta encoding. This includes skip delta-based VCS systems.

C. RDC–AVCS: An Auditable Version Control System based on Remote Data Checking

In this section, we present our main result, RDC–AVCS, the first auditable version control system. RDC–AVCS is obtained

[3]We assume that an (empty) VCS repository has been already created, *e.g.*, by using the SVN command "`svnadmin create`".

by integrating RDC mechanisms into a VCS system. Whereas our definition of AVCS targets VCS systems that use delta encoding in general, in our RDC–AVCS construction we focus on VCS systems that use skip delta-based encoding. As explained in Sec. II-A, these are optimized for both storage and retrieval; however, they are arguably more challenging to secure than VCS systems that use delta encoding, because of the nature of computing the skip deltas.

Challenges. Going from a benign setting to an adversarial setting, we need to overcome several challenges. These challenges stem from the adversarial nature of the VCS server and from the format of a skip delta-based VCS repository which is optimized to minimize the server's storage and workload during the Retrieve phase:

– *The gap between the server's and the client's view of the repository.* In a general-purpose RDC protocol (Sec. II-B), the client and the server have the same view of the outsourced data: the client computes the verification metadata based on the data, and then sends both data and metadata to the server. The server stores these unmodified. The server then uses the data and metadata to answer the client's challenges by computing a proof that convinces the client that the server continues to store the same data outsourced by the client.

However, in a skip delta-based VCS, there is a gap between the two views, which makes skip delta-based VCS systems more difficult to audit: Although both client and server view the main VCS repository as the initial version of the data plus a series of delta files corresponding to subsequent data versions, they have a different understanding of the delta files. To commit a new version t, the client computes and sends to the server a delta that is the difference between the new version and its immediate previous version, that is the difference between version t and $t-1$ (recall that the client only stores the working copy which is version $t-1$, and version t which incorporates the changes made by the client over version $t-1$). However, this is different than the skip deltas that are stored by the server: a δ_i file stored by the server is the difference between version i and a "skip version", which is not necessarily the immediate version previous to i. For example, the skip delta for version 128 will be computed as the difference against version 0 (the algorithm for selecting the "skip version" is described in Sec. II-A). Since the client does not have access to the skip deltas stored by the server, it cannot compute the verification metadata over them, as needed in an RDC protocol.

– *Delta encoding is not reversible.* The client may try to retrieve the skip delta computed by the server and then compute the verification metadata based on the retrieved skip delta. However, in an adversarial setting, the client cannot trust the server to provide a correct skip delta value. This is exacerbated by the fact that delta encoding is not a reversible operation. If $\delta_{t-1\rightarrow t}$ is the difference between versions $t-1$ and t (*i.e.*, $\mathtt{F}_t = \mathtt{F}_{t-1} + \delta_{t-1\rightarrow t}$), this does not imply that $\mathtt{F}_{t-1}$ can be obtained based on $\mathtt{F}_t$ and $\delta_{t-1\rightarrow t}$. The reason comes from the method used by delta encoding to encode update operations between versions, such as insert, update, delete. If a delete operation was executed on version $t-1$ to obtain version t, then $\delta_{t-1\rightarrow t}$ encodes only the position of the deleted portion from $\mathtt{F}_{t-1}$, so that given $\mathtt{F}_{t-1}$ and $\delta_{t-1\rightarrow t}$, one can obtain $\mathtt{F}_t$. However, $\delta_{t-1\rightarrow t}$ does not encode the actual data that has been deleted. Thus, $\mathtt{F}_{t-1}$ cannot be obtained based on $\mathtt{F}_t$ and $\delta_{t-1\rightarrow t}$.

A first attempt. We make two observations which we then leverage to build an initial, alas inefficient AVCS system:

– *First*, we observe that any RDC protocol that supports the append operation securely can be used to audit the integrity of a VCS server that relies on skip delta encoding, simply because RDC can be used to spot check the blocks of a virtual file obtained by concatenating the original file and the subsequent delta files. In Sec. V-A, we show that existing RDC protocols proposed for static data can be enhanced to securely support the append operation.

– *Second*, we need to unify the client's and server's views of the repository data so that the client can compute on its own the metadata over the delta files that are stored at the server.

To bridge the gap between the server's and the client's view of the repository, we require that, upon each commit, the skip delta is computed by the client and not by the server. The client will then send the skip delta to the server, together with RDC verification tags computed over the skip delta. To be able to compute the skip delta, the client should store several previous versions, so that it has access to the "skip version" against which the skip delta is computed. Our analysis in Appendix B shows that, unfortunately, the storage required for storing enough previous versions on the client side is linear with the total number of versions in a repository. This does not conform with our notion of outsourcing the VCS repository, in which the client should only store one version of the file (the working copy).

1) The RDC–AVCS Construction: We are now ready to present RDC–AVCS, an auditable VCS scheme which uses RDC mechanisms to ensure all the versions of a file can be retrieved from the VCS server. RDC–AVCS only requires the same amount of storage on the client like a regular VCS system. This scheme is the main result of the paper.

Recall that the VCS repository contains t versions of the file, $\mathtt{F}_0, \mathtt{F}_1, ..., \mathtt{F}_{t-1}$. The t versions are stored in the repository as t files: $\mathtt{F}_0$, δ_1, δ_2, ..., δ_{t-1} (*i.e.*, the initial version of the file and $t-1$ skip delta files).

For the purpose of our scheme, we view all the information pertaining to the versions of the file $\mathtt{F}$ as a virtual file $\tilde{\mathtt{F}}$ obtained by concatenating the original file and the subsequent delta files: $\tilde{\mathtt{F}} = \mathtt{F}_0||\delta_1||\delta_2||...\delta_{t-1}$. We view $\tilde{\mathtt{F}}$ as a collection of fixed-size blocks, each block containing s symbols, and each symbol is an element of $GF(p)$, where p is a large prime (at least 80 bits). This view matches the view of a file in an RDC scheme: To check the integrity of all the versions of $\mathtt{F}$, it is enough to check the integrity of $\tilde{\mathtt{F}}$. Let n denote the number of blocks in $\tilde{\mathtt{F}}$. As the client commits new file versions, n will grow accordingly (note that n is maintained by the client).

RDC–AVCS **overview.** We use two types of verification tags. To check data possession (in the Challenge phase) we use *challenge tags*; these are computed over the blocks in $\tilde{\mathtt{F}}$ to facilitate spot checking in RDC [26]. To check the integrity of individual file versions (in both the Commit and the Retrieve

phases), we use *retrieve tags*; these are computed over entire versions of $\mathtt{F}$.

To check the integrity of $\tilde{\mathtt{F}}$, we adopt the challenge tags introduced by Shacham and Waters [46][4]. When the client commits a new file version, it computes a retrieve tag in the form of a MAC over the whole file version that is to be committed. This retrieve tag will be stored at the VCS server and will be used by the server to convince the client of file version integrity during Commit and Retrieve.

In a benign setting, whenever the client commits a new file version, the server computes and stores a skip delta file in the main VCS repository (as described in Sec. IV-A). Under an adversarial setting, to leverage RDC techniques over the VCS repository, the skip delta files must be accompanied by verification challenge tags. Since the challenge tags can only be computed by the client, our scheme requires the client to obtain the skip delta, compute the challenge tags over it and send both the skip delta and the tags to the server.

When committing a new version $\mathtt{F}_t$, the client must compute the skip delta (δ_{skip}) for $\mathtt{F}_t$. The δ_{skip} must be computed against a certain previous version of the file, called the "skip version" (as described in Sec. II-A). Recall that the client also has in its local store a copy of $\mathtt{F}_{t-1}$, the working copy.

If $(skip(t) == t-1)$, then the client can directly compute δ_{skip} such that $\mathtt{F}_t = \mathtt{F}_{t-1} + \delta_{skip}$. Otherwise, the client computes δ_{skip} by interacting with the VCS server as follows:

1. The client computes the difference between the new version and the immediate previous version, *i.e.* computes δ such that $\mathtt{F}_t = \mathtt{F}_{t-1} + \delta$. The client sends δ to the server.
2. The server re-computes $\mathtt{F}_{t-1}$ based on the data in the repository and then computes $\mathtt{F}_t = \mathtt{F}_{t-1} + \delta$. The server then re-computes $\mathtt{F}_{skip(t)}$ (the skip version for $\mathtt{F}_t$) based on the data in the repository and computes the difference between $\mathtt{F}_t$ and $\mathtt{F}_{skip(t)}$, *i.e.* it computes $\delta_{reverse}$ such that $\mathtt{F}_{skip(t)} = \mathtt{F}_t + \delta_{reverse}$. The server sends $\delta_{reverse}$ to the client, together with the retrieve tag for $\mathtt{F}_{skip(t)}$.
3. The client computes the skip version: $\mathtt{F}_{skip(t)} = \mathtt{F}_t + \delta_{reverse}$ and checks the validity of $\mathtt{F}_{skip(t)}$ using the retrieve tag received from the server. The client then computes the skip delta for the new file version, *i.e.* δ_{skip} such that $\mathtt{F}_t = \mathtt{F}_{skip(t)} + \delta_{skip}$.

To give an example, when the client commits $\mathtt{F}_{15}$, the client also has the working copy $\mathtt{F}_{14}$ which is the skip version for $\mathtt{F}_{15}$, and the client can compute directly δ_{skip} such that $\mathtt{F}_{15} = \mathtt{F}_{14} + \delta_{skip}$. However, when the client commits $\mathtt{F}_{20}$, it only has $\mathtt{F}_{19}$ in her local store and must first retrieve from the server $\delta_{reverse}$ and then compute $\mathtt{F}_{16}$ which is the skip version for $\mathtt{F}_{20}$, as $\mathtt{F}_{16} = \mathtt{F}_{20} + \delta_{reverse}$. Only then can the client compute δ_{skip} such that $\mathtt{F}_{20} = \mathtt{F}_{16} + \delta_{skip}$.

For the Challenge phase, we leverage a mechanism based on checking the integrity of the remotely stored data, like in previous RDC schemes [26], [43]. With every challenge, the client challenges the server to prove possession of a random subset of the blocks in $\tilde{\mathtt{F}}$. The server provides a proof of possession which convinces the client that the server can produce the data in the challenged blocks. This spot checking mechanism is quite efficient. For example, when the server corrupts 1% of the repository (*i.e.*, 1% of $\tilde{\mathtt{F}}$), then the client can detect this corruption with high probability by randomly checking only a small constant number of blocks (*e.g.*, checking 460 blocks results in a 99% detection probability) [26].

In the Retrieve phase, the client replaces her working copy with another file version. The client can use the corresponding retrieve tag to check the correctness of the file version provided by the server.

The RDC–AVCS scheme. The details of the RDC–AVCS scheme are presented in Figures 2, 3 and 4. Let $\tilde{\mathtt{F}}$ be a virtual file obtained by concatenating the original file and the subsequent delta files: $\tilde{\mathtt{F}} = \mathtt{F}_0||\delta_1||\delta_2||...\delta_{t-1}$. Let n be the number of blocks in $\tilde{\mathtt{F}}$. The client maintains n and updates n accordingly whenever she commits a new file version to the repository.

The Setup phase. The client runs KeyGen to generate two private keys K_1 and K_2, and picks s random numbers from $GF(p)$, which will be used in computing the challenge tags. The client also sets $n = 0$.

The Commit phase. To commit a new file version, the client uses ComputeDelta to compute the skip delta for the new file version, and runs GenMetadata to generate the corresponding challenge and retrieve tags. In ComputeDelta (Fig. 3), the client first uses SelectSkipVersion to determine the skip version. If the skip version is the immediate previous version of the new version, the client simply computes the skip delta based on the new version and its immediate previous version. Otherwise, the client contacts the server, sending the delta of the new version against its immediate previous version. The server uses ComputeReverseAndSkipDelta to generate the delta of the skip version against the new version, *i.e.*, $\delta_{reverse}$, and returns to the client $\delta_{reverse}$ and the retrieve tag of the skip version. The client then re-computes the skip version based on the new version and $\delta_{reverse}$, and verifies the validity of the computed skip version by running CheckRetrieveProof. If the verification succeeds, the client computes the skip delta based on the new version and the skip version. After having computed the skip delta, the client runs GenMetadata (Fig. 3) to compute the challenge tags and the retrieve tag, which will then be sent to the server. The retrieve tag $\mathtt{R}_t$ is computed using an HMAC function [44]. Finally, the client increases n by d, where d is the number of blocks in the skip delta.

The Challenge phase. Periodically, the client challenges the server to prove possession of the virtual file $\tilde{\mathtt{F}}$. The client sends a challenge to the server, in which it selects a random subset of c blocks for checking. The server runs GenProof to generate the corresponding proof, and sends it back to the client. The client then checks the validity of the received proof by running CheckProof.

The Retrieve phase. The Retrieve phase is activated when the client wants to replace her working copy with an older or a newer version. The client sends a request to the server.

[4]For efficiency reasons, we use the tags that support private verifiability. However, our scheme could also be instantiated using the challenge tags in [46] that are publicly verifiable.

Let κ be a security parameter. Let $h : \{0,1\}^\kappa \times \{0,1\}^* \rightarrow GF(p)$ be a PRF. All arithmetic operations are over the field $GF(p)$ of integers modulo p, where p is a large prime (at least 80 bits), unless noted otherwise explicitly. RDC–AVCS has four phases: Setup, Commit, Challenge, and Retrieve.

Setup: The client runs $(K_1, K_2) \leftarrow \mathsf{KeyGen}(1^\kappa)$ and picks s random numbers $\alpha_1, \ldots, \alpha_s$ from $GF(p)$. The client sets $n = 0$

Commit: Having made updates to her working copy $\mathtt{F}_{t-1}$, the client C wants to commit to the repository a new version $\mathtt{F}_t$. C performs the following operations:

1. Compute δ for $\mathtt{F}_t$ against the immediate previous version $\mathtt{F}_{t-1}$, such that $\mathtt{F}_t = \mathtt{F}_{t-1} + \delta$
2. Run $(\delta_{skip}, skip(t)) \leftarrow \mathsf{ComputeDelta}(K_2, \delta, t, \mathtt{F}_t)$
3. View δ_{skip} as a collection of blocks and run $(\mathtt{R}_t, \mathtt{T}_{begin}, \ldots, \mathtt{T}_{end}, begin, end) \leftarrow \mathsf{GenMetadata}(K_1, K_2, \delta_{skip}, n, \alpha_1, \ldots, \alpha_s, \mathtt{F}_t, t)$. This computes a set of challenge tags $\{\mathtt{T}_{begin}, \ldots, \mathtt{T}_{end}\}$ for the blocks in δ_{skip} and a retrieve tag $\mathtt{R}_t$ for $\mathtt{F}_t$.
4. If $(skip(t) == t-1)$ then send $(\delta, \mathtt{T}_{begin}, \ldots, \mathtt{T}_{end}, \mathtt{R}_t)$ to server S; Otherwise, send $(\mathtt{T}_{begin}, \ldots, \mathtt{T}_{end}, \mathtt{R}_t)$ to S
5. Update the number of blocks in $\tilde{\mathtt{F}}$: $n = end$

Challenge: Client C uses spot checking to check possession of the virtual file $\tilde{\mathtt{F}}$. In this process, the server S uses its stored repository and the corresponding challenge tags to prove data possession.

1. C generates a challenge Q and sends Q to S. The challenge Q is a c-element set $\{(j, v_j)\}$, in which j denotes the index of the block in $\tilde{\mathtt{F}}$ to be challenged, and v_j is chosen at random from $GF(p)$.
2. S runs $(\mu_1, \ldots, \mu_s, \sigma) \leftarrow \mathsf{GenProof}(Q, \tilde{\mathtt{F}}, \mathtt{T}_1, \ldots, \mathtt{T}_n)$ and returns to C the proof of possession $(\mu_1, \ldots, \mu_s, \sigma)$
3. C checks the validity of the proof $(\mu_1, \ldots, \mu_s, \sigma)$ by running $\mathsf{CheckProof}(K_1, \alpha_1, \ldots, \alpha_s, Q, \mu_1, \ldots, \mu_s, \sigma)$

Retrieve: To replace version j (the working copy) with another version i, the client C executes:

1. C sends a request to the server S
2. The server S runs $(\delta_{retrieve}, \mathtt{R}_i) \leftarrow \mathsf{GenRetrieveVersionAndProof}(j, i)$ and returns to the client $\delta_{retrieve}$ and the retrieve tag $\mathtt{R}_i$ for version i
3. C computes $\mathtt{F}_i$: $\mathtt{F}_i = \mathtt{F}_j + \delta_{retrieve}$
4. C checks the validity of $\mathtt{F}_i$ by running $\mathsf{CheckRetrieveProof}(K_2, \mathtt{F}_i, i, \mathtt{R}_i)$

Fig. 2: The RDC–AVCS system.

The server uses GenRetrieveVersionAndProof to generate the delta of the desired file version against the client's local version ($\delta_{retrieve}$ in Fig. 2), together with the retrieve tag of the desired file version. Both the delta and the retrieve tag are returned to the client. The client then computes the desired file version, and checks its validity by running CheckRetrieveProof.

V. Analysis and Discussion

A. Security Analysis

The security of the RDC–AVCS scheme is captured by the following lemmas and theorems:

Lemma V.1 (Corruption Detection Guarantee)**.** *Assume that*

$\mathsf{KeyGen}(1^\kappa)$: Choose two keys K_1, K_2 at random from $\{0,1\}^\kappa$.
Return (K_1, K_2)

$\mathsf{ComputeDelta}(K_2, \delta, t, \mathtt{F}_t)$:

1. Initialize the skip delta for $\mathtt{F}_t$: $\delta_{skip} = \delta$
2. Run $skip(t) \leftarrow \mathsf{SelectSkipVersion}(t)$
3. If $(skip(t) \neq t-1)$ then client C executes:
   (a) Send $(\delta, t, skip(t))$ to the server S
   (b) The server S runs $(\delta_{reverse}, \delta_{skip}) \leftarrow \mathsf{ComputeReverseAndSkipDelta}(\delta, t, skip(t))$. S stores δ_{skip} and sends $(\delta_{reverse}, \mathtt{R}_{skip(t)})$ back to C
   (c) The client C re-computes $\mathtt{F}_{skip(t)}$: $\mathtt{F}_{skip(t)} = \mathtt{F}_t + \delta_{reverse}$. C runs $\mathsf{CheckRetrieveProof}(K_2, \mathtt{F}_{skip(t)}, skip(t), \mathtt{R}_{skip(t)})$ to check the correctness of the $\delta_{reverse}$ received from S. If the check fails, conclude that S is faulty and exit. Otherwise, compute δ_{skip} for $\mathtt{F}_t$, such that $\mathtt{F}_t = \mathtt{F}_{skip(t)} + \delta_{skip}$
4. Return $(\delta_{skip}, skip(t))$

$\mathsf{GenMetadata}(K_1, K_2, \delta, n, \alpha_1, \ldots, \alpha_s, \mathtt{F}_t, t)$:

1. $begin = n + 1$
2. View δ as a collection of d fixed-size blocks: $\delta = (\mathtt{b}_{n+1}, \ldots, \mathtt{b}_{n+d})$. For the purpose of computing challenge tags, we use the range $[n+1, n+d]$ for the block indices of the blocks in δ. Each block $\mathtt{b}_i$ in δ contains s symbols from $GF(p)$: $\mathtt{b}_i = (\mathtt{b}_{i,1}, \ldots, \mathtt{b}_{i,s})$.
3. $end = n + d$
4. For $begin \leq j \leq end$: $\mathtt{T}_j = h_{K_1}(j) + \sum_{k=1}^{s} \alpha_k \mathtt{b}_{jk}$
5. $\mathtt{R}_t = HMAC_{K_2}(\mathtt{F}_t || t)$
6. Return $(\mathtt{R}_t, \mathtt{T}_{begin}, \ldots, \mathtt{T}_{end}, begin, end)$

$\mathsf{GenProof}(Q, \tilde{\mathtt{F}}, \mathtt{T}_1, \ldots, \mathtt{T}_n)$:

1. Parse Q as a set of c pairs (j, v_j). Parse $\tilde{\mathtt{F}}$ as $\{\mathtt{b}_1, \ldots, \mathtt{b}_n\}$.
2. Compute the proof of possession $(\mu_1, \ldots, \mu_s, \sigma)$:
   - For $1 \leq k \leq s$: $\mu_k = \sum_{(j,v_j) \in Q} v_j \mathtt{b}_{jk} \mod p$
   - $\sigma = \sum_{(j,v_j) \in Q} v_j \mathtt{T}_j \mod p$
3. Return $(\mu_1, \ldots, \mu_s, \sigma)$

$\mathsf{CheckProof}(K_1, \alpha_1, \ldots, \alpha_s, Q, \mu_1, \ldots, \mu_s, \sigma)$:

1. Parse Q as a set of c pairs (j, v_j)
2. If $\sigma = \sum_{(j,v_j) \in Q} v_j h_{K_1}(j) + \sum_{k=1}^{s} \alpha_k \mu_k \mod p$, return "success". Otherwise return "failure".

$\mathsf{GenRetrieveVersionAndProof}(j, i)$:

1. Compute $\mathtt{F}_j$ by starting from $\mathtt{F}_0$ and apply the corresponding skip deltas
2. Compute $\mathtt{F}_i$ by starting from $\mathtt{F}_0$ and apply the corresponding skip deltas
3. Compute $\delta_{retrieve}$ such that $\mathtt{F}_i = \mathtt{F}_j + \delta_{retrieve}$
4. Get the retrieve tag $\mathtt{R}_i$ from the repository
5. Return $(\delta_{retrieve}, \mathtt{R}_i)$

$\mathsf{CheckRetrieveProof}(K_2, \mathtt{F}_t, t, \mathtt{R})$:

1. $\mathtt{R}_t = HMAC_{K_2}(\mathtt{F}_t || t)$
2. if $(\mathtt{R}_t == \mathtt{R})$ then return true; Otherwise, return false

Fig. 3: The RDC–AVCS scheme.

SelectSkipVersion(t):

1. Considering the binary representation of the version number t, obtain $skip(t)$ by changing the rightmost bit that has value "1" into a bit with value "0"
2. Return $skip(t)$

ComputeReverseAndSkipDelta$(\delta, t, skip(t))$:

1. Retrieve $\mathtt{F}_t$'s immediate previous version, $\mathtt{F}_{t-1}$, based on the data in the repository
2. Compute $\mathtt{F}_t$: $\mathtt{F}_t = \mathtt{F}_{t-1} + \delta$
3. Retrieve $\mathtt{F}_{skip(t)}$ based on the data in the repository
4. Compute $\delta_{reverse}$, such that $\mathtt{F}_{skip(t)} = \mathtt{F}_t + \delta_{reverse}$
5. Compute the skip delta δ_{skip} for $\mathtt{F}_t$, such that $\mathtt{F}_t = \mathtt{F}_{skip(t)} + \delta_{skip}$
6. Return $(\delta_{reverse}, \delta_{skip})$

Fig. 4: Components of the RDC–AVCS scheme.

the server stores an n-block file, out of which x blocks are corrupted. By randomly checking c different blocks over the entire file, the verifier (client) will detect the corruption with probability at least $1 - (1 - \frac{x}{n})^c$.

Proof: We refer the reader to [26], [25] for the proof. ■

Based on Lemma V.1, if the server corrupts 1% of the whole file then, by randomly checking 460 blocks, the verifier can detect the corruption with a probability of at least 99%, regardless of the file size.

Lemma V.2. *Let S be an RDC scheme, designed for static data, which achieves the PDP security guarantee for a file $\mathtt{F}$ outsourced at un untrusted third party [26], [25], and let S' be another RDC scheme obtained by enhancing S to support the append operation: Blocks can be appended at the end of $\mathtt{F}$ and for each appended block a verification tag is computed by the client and stored at the server. Then S' also achieves the PDP security guarantee for the updated file.*

Proof: (sketch) We show that an RDC scheme can guarantee data possession of an updated version of the file after an arbitrary number of appends are performed. Assume the client outsources a file $\mathtt{F}$, which has n blocks $\mathtt{b}_1, \mathtt{b}_2, \ldots, \mathtt{b}_n$. The client applies RDC scheme S over this file as follows. During the Setup phase, it computes verification tags $\mathtt{T}_1, \mathtt{T}_2, \ldots, \mathtt{T}_n$ for all the blocks in $\mathtt{F}$. The verification tag $\mathtt{T}_i$ is computed over the data in file block $\mathtt{b}_i$ and also over i, the index of block $\mathtt{b}_i$ in $\mathtt{F}$. The client then outsources $\mathtt{F}$ as well as the verification tags to the untrusted server. During the Challenge phase, the verifier (client) uses spot checking to check the integrity of $\mathtt{F}$ [26]. This RDC scheme S guarantees data possession of file $\mathtt{F}$. We obtain a new RDC scheme S' from S by adding support for the append operation. When the client wants to append a new block $\mathtt{b}_{n+1}$ to file $\mathtt{F}$, the client computes a new verification tag $\mathtt{T}_{n+1}$ over the data in $\mathtt{b}_{n+1}$ and over the index $n+1$ of the new block. The client then sends $\mathtt{b}_{n+1}$ and $\mathtt{T}_{n+1}$ to the server. From the client's view, the server should now store the new file $\mathtt{F}'$, which has $n+1$ blocks $\mathtt{b}_1, \mathtt{b}_2, \ldots, \mathtt{b}_n, \mathtt{b}_{n+1}$, together with the set of tags $\mathtt{T}_1, \mathtt{T}_2, \ldots, \mathtt{T}_n, \mathtt{T}_{n+1}$. The same argument used to prove that S achieves the PDP security guarantee over the initial file $\mathtt{F}$ can now be used to show that S' achieves the PDP security guarantee over the updated file $\mathtt{F}'$. By induction, S' can guarantee data possession of any updated version of the file after an arbitrary number of append operations are performed. Thus, we conclude that a PDP scheme which supports the append operation can achieve the PDP security guarantee for the updated file. ■

Lemma V.3. RDC–AVCS *guarantees that skip delta files are correctly computed by the client.*

Proof: (sketch) The skip delta may be computed in two ways during the Commit phase:

– The skip version is the version immediately previous to the new version ($skip(t) = t-1$). In this case, the client computes directly the correct skip delta.

– The skip version is not the version immediately previous to the new version ($skip(t) \neq t-1$). In this case, the client cooperates with the untrusted server to compute the skip delta. The client computes the skip version of the file based on the data received from the server and then verifies the correctness of the skip version using the retrieve tag provided by the server. This check guarantees the correctness of the skip version, since the retrieve tag was previously computed by the client. If this check is successful, the client then computes the correct skip delta.

In both cases, the skip delta is guaranteed to be correctly computed by the client. ■

Lemma V.3 guarantees that the client computes challenge tags over the correct skip deltas. This is important, because otherwise corruptions introduced during the commit operation may go undetected and may get incorporated in the VCS repository.

Theorem V.4. RDC–AVCS *achieves security guarantees **SG1** and **SG2**.*

Proof: (sketch). In RDC–AVCS, the repository, which is the collection of t versions of file $\mathtt{F}$, can be seen as a virtual file $\tilde{\mathtt{F}}$, obtained by concatenating the initial file version $\mathtt{F}_0$, and the skip delta files δ_1, ..., δ_{t-1} corresponding to the subsequent versions. In this view, committing a new version to the repository is equivalent to appending the corresponding skip delta to the file $\tilde{\mathtt{F}}$. During the Commit phase, when committing the initial file version $\mathtt{F}_0$, the client computes the challenge tags over $\mathtt{F}_0$, and when committing each subsequent version, the client computes the challenge tags over the corresponding skip delta as if the skip delta is appended to $\tilde{\mathtt{F}}$. According to Lemma V.3, each skip delta is guaranteed to be correctly computed by the client.

During the Challenge phase, the client uses spot checking to check the integrity of $\tilde{\mathtt{F}}$. RDC schemes for static data, in which there is a verification tag for each file block have been shown to achieve the PDP security guarantee [26], [46], *i.e.*, the client can detect corruption of a fraction of the outsourced data. RDC–AVCS falls in the same category, except it supports an additional operation, append to $\tilde{\mathtt{F}}$. According to lemma V.2, an RDC scheme supporting append operation achieves the same security guarantee as an RDC scheme for static data. Finally, according to lemma V.1, the verifier in RDC can detect if the server corrupts a fraction of the outsourced file; thus, our RDC–AVCS scheme achieves the security guarantee **SG1**.

In RDC–AVCS, the client computes a retrieve tag for each file version $\mathtt{F}_i$ by applying an HMAC over the concatenation of the file version content ($\mathtt{F}_i$) and the version number (i) using a secret key (K_2). The security of HMAC guarantees that the adversary cannot forge a retrieve tag without knowing the secret key. Furthermore, the adversary cannot perform a replay attack by providing in the Retrieve phase a different file version than the one requested by the client. We conclude that the RDC–AVCS client can verify the correctness of the retrieved versions, thus achieving the security guarantee **SG2**. ■

B. Performance Analysis

During the Commit phase, the client interacts with the sever to compute the skip deltas. To retrieve any file version from the repository, the server has to go through at most $log(t)$ skip deltas, thus, the server computation is $O(nlog(t))$. The client has to compute the skip version and the skip delta, and generate the metadata, which require a computation complexity linear in the version size (see Table I). The communication in a commit operation is also linear with the version size, since it mainly includes two deltas (Figure 2 and 3) and a set of challenge tags for a skip delta.

During the Challenge phase, RDC–AVCS adopts the spot checking technique, in which the client challenges the server to prove possession of a random subset of the blocks in $\tilde{\mathtt{F}}$ (the number of challenged blocks is always a small constant [25]), and the server generates a proof of data possession by aggregating the selected blocks and the corresponding challenge tags. Thus, the computation (client and server) and the communication complexity are both $O(1)$ (Table I). This is a major advantage of RDC–AVCS compared to previous schemes, in which the checking complexity is determined either by the repository size or the version size (Table I).

During the Retrieve phase, to retrieve a version from the repository, the server needs to apply at most $log(t)$ skip deltas, thus, the server computation is $O(nlog(t))$. Previous schemes which are built on top of delta encoding (or can be easily built on top of delta encoding) impose $O(nt)$ computation on the server (Table I). The client storage overhead in RDC–AVCS is $O(n)$, since the client always stores locally the working copy.

C. Remarks

Small corruption protection. In RDC–AVCS, we adopt spot checking during the Challenge phase for efficiency reasons. Spot checking was shown to detect data corruption with high probability if the server corrupts a fraction of the data [25]. This provides defense against an adversary which is rational and economically motivated, *i.e.*, one that will not cheat unless it can achieve a clear financial gain without being detected. However, spot checking is not necessarily effective under a stronger adversary, *e.g.*, an adversary which is fully malicious. Spot checking cannot detect if the adversary corrupts a small amount of the data, such as 1 byte. To provide protection against small amounts of data corruption – a property called *robustness* – previous RDC schemes for static data rely on a special application of error correcting codes to generate redundant data, so that small corruptions that are not detected can be repaired [26], [31], [30]. Integrating error correcting codes with RDC when dynamic updates can be performed on the data is much more challenging than in the static setting. A few RDC solutions have been proposed to achieve robustness for the dynamic setting, but this involves substantial additional cost: one system requires to store a large amount of redundant data on the client side [47]; other systems store and access the redundant data on the server side either by requiring the client to access the entire redundancy [35] or by using inefficient mechanisms such as PIR that hide the access pattern [33].

In this work, we choose to sacrifice robustness for two reasons. First, the solutions proposed to achieve robustness for RDC under a dynamic setting are designed to handle the full range of update operations (insertions, deletions, modifications) and are thus overkill for version control systems where the only meaningful operation is append. Second, one of our main design goals was to achieve an auditable VCS scheme which is efficient and has performance comparable to a regular (non-secure) VCS system.

Multiple-file support. We have described RDC–AVCS for the case when the main repository only contains the versions of one file. A challenge tag for block with index j in $\tilde{\mathtt{F}}$ is computed as $\mathtt{T}_j = h_{K_1}(j) + \sum_{k=1}^{s} \alpha_k \mathtt{b}_{jk}$. The index j used in the challenge tag should be different across all the challenge tags. In other words, the client should not reuse the same index j twice for computing challenge tags. In this case, the index j used in the challenge tag is the block's position in the file $\tilde{\mathtt{F}}$, which ensures its unicity. When multiple files are stored in the VCS repository, the client must ensure that the indices used to compute the challenge tags are different not only across blocks of the same file, but also across blocks of different files. This could be achieved by prepending a file identifier to the block index. For example, if the identifier of a file $\mathtt{F}$ is given by $id(\mathtt{F})$ and assuming that each file has a unique identifier, then for the blocks in the various versions of $\mathtt{F}$, the client computes challenge tags as $\mathtt{T}_j = h_{K_1}(id(\mathtt{F})||j) + \sum_{k=1}^{s} \alpha_k \mathtt{b}_{jk}$. Similarly, the file's identifier should be embedded in the retrieve tag for version $\mathtt{F}_i$: $\mathtt{R}_i = h_{K_2}(\mathtt{F}_i||id(\mathtt{F})||i)$.

VI. Implementation and Experiments

A. Implementation

We built a prototype for RDC–AVCS on top of Apache Subversion (SVN) [4], a popular open-source version control system. We added about 4,000 lines of C code into the SVN code base (V1.7.8), and built Secure SVN (SSVN), a secure version control system based on skip delta encoding. Since many SVN repositories already exist, we also built a tool, SSVN-Migrate, which converts an existing (non-secure) SVN repository into a SSVN repository.

Implementation overview. We modified the source code in both SVN client and SVN server. For the SVN client, we mainly modified the following SVN commands

– svn add: add files to the working copy. The corresponding new command in SSVN is "ssvn add".

– svn rm: remove files from the working copy. The corresponding new command in SSVN is "ssvn rm".

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

– svn co: checkout the latest version of the data. The corresponding new command in SSVN is "ssvn co".

– svn update: update the current version to an arbitrary version. The corresponding new command in SSVN is "ssvn update".

For the SVN server, we modified the stand-alone server "svnserve". The new server is named "sec-svnserve".

During the Commit phase, the client updates the working copy and wants to commit the changes to the repository. In RDC–AVCS, the changes for a new version $\mathtt{F}_t$ are encoded in the skip delta, δ_{skip}, which is the difference between the skip version and the new version, *i.e.*, $\mathtt{F}_t = \mathtt{F}_{skip(t)} + \delta_{skip}$. The algorithm for computing δ_{skip} is described in Sec. IV-C1. After computing the skip delta, the client computes the set of challenge tags for it and a retrieve tag for the new version, and sends them to the server.

In SSVN we added functionality to the original SVN client ("svn commit"), so that the SSVN client ("ssvn commit") can communicate with the server to compute the skip delta, as well as compute the challenge and retrieve tags. We also added functionality to the original SVN server ("svnserve") to allow the server to compute and send back the delta of skip version against the new version, together with a proof for checking the validity of the skip version.

During the Retrieve phase, the client wants to revert the working copy to an older version or update it to a newer version. It sends a request to the server, which retrieves the requested version from the repository, together with the corresponding retrieve tag. The server can then choose to send back either the whole requested version or the delta between the requested version and the working copy (SVN uses the latter strategy). The client further validates the requested version based on the retrieve tag. Correspondingly, in SSVN we added additional functionality to the original SVN client ("svn co" and "svn update"), so that the SSVN client ("ssvn co" and "ssvn update") can verify the retrieve tags for the affected files. We also added additional functionality to the original SVN server ("svnserve") to allow it to retrieve and send back the corresponding retrieve tags for the affected files.

Implementation issues. We highlight next some of the most interesting implementation issues we encountered. First, we had to bridge the gap between how RDC–AVCS and SVN view the data: RDC–AVCS abstracts each version of the data as a file, and thus one simply performs update operations to this file. However, in SVN, each version is associated with a project, which is a collection of files, and the delta (*i.e.*, skip delta) is computed independently for each file. In addition, files can be added and deleted from the project. To reconcile the different views, we apply RDC–AVCS over each file in an SVN project, *i.e.*, we have a virtual project for each file, and the SVN project is a collection of virtual projects corresponding to the files in the SVN project. When a file is added to the project, the corresponding virtual project is initialized; when this file is updated (*i.e.*, insert, delete, modify, or append data), the corresponding virtual project is updated; when the file is deleted, the corresponding virtual project should be kept rather than be deleted.

Another implementation issue is related to how SVN handles memory management. Rather than requesting memory directly from the OS using the standard $malloc()$ function, SVN relies on Apache Portable Runtime (APR) [2] library for memory management. Specifically, a program that links against APR can request a pool of memory by using $apr\_pool\_create()$, and APR will allocate a moderate-size chunk of memory from the OS which will be available for use to the program immediately. The pool will automatically grow in size to accommodate programs that request more memory than the original pool contained. Unfortunately, without carefully reclaiming back memory from the pool when handling a large number of files, the pool becomes full, leading to an "out of memory" error. In SSVN, we tackled this issue by clearing the pool after having handled a certain number of files, *e.g.*, 1000. We tested that SSVN is robust enough to handle hundreds of thousands of files in a single commit operation.

B. Experimental Setup

We ran experiments in which both the server and the client are running on the same machine, an Intel Core 2 Duo system with two CPUs (each running at 3.0GHz, with a 6144KB cache), 1.333GHz frontside bus, 4GB RAM and a Hitachi HDP725032GLA360 360GB hard disk with ext4 file system. The system runs Ubuntu 12.10, kernel version 3.5.0-17-generic. We used the OpenSSL library [16] version 1.0.1e.

Repository selection. We categorized the existing SVN repositories into three groups based on the number of files in the repository: A small-size repository has less than $5,000$ files, a medium-size repository has between $5,000$ and $50,000$ files, and a large-size repository has more than $50,000$ files. Based on these criteria, we selected three representative public SVN repositories for our experimental evaluation: FileZilla [9] for small-size repository, Wireshark [23] for medium-size repository, and GCC [11] for large-size repository. Table II shows statistics about these three repositories.

	FileZilla	Wireshark	GCC
Dates of activity	2001-2013	1998-2013	1987-2013
Number of versions	5,119	49,946	200,127
Number of files	1,023	5,342	80,183
Average filesize	19KB	32KB	6KB
Repository category	small size	medium size	large size

TABLE II: Statistics for the selected repositories (as of June 2013). The number of files and the average filesize is estimated based on the latest version in the repository.

Overview of experiments. We evaluated the computation and communication overhead during the Commit phase (Sec. VI-C) and the computation overhead during the Retrieve phase (Sec. VI-D), for both SSVN and SVN. The Challenge phase has been shown to be very efficient for RDC schemes which rely on spot checking [25], so we do not include it in our experiments.

We average the overhead over the first 1000 versions of the three repositories (labeled FileZilla, Wireshark and GCC1).

GCC has a large-size repository, with more than 200K versions and more than 80K files in its latest version. Since for GCC the difference between the first 1000 versions and the last 1000 versions is considerable in the size of the repository, we also included in our experiments an average of the overhead over the last 1000 versions of GCC (labeled GCC2).

In Sec. VI-E, we describe the migration tool which seamlessly converts an existing (non-secure) SVN repository to a SSVN repository; we also perform an experiment in which we migrate the first 3000 versions of the aforementioned three repositories.

C. Commit Phase

For SVN and SSVN, we evaluated the computation and communication overhead for the commit operation. To measure the time for a commit operation, we measured the time needed for running the shell commands "svn commit" and "ssvn commit" to commit a version. To measure the communication overhead of non-secure SVN for a commit operation, we observed that the non-secure SVN client relies on two write functions $writebuf\_output$ and $svn\_ra\_svn\_writebuf\_output$ to send data, and two read functions $readbuf\_input$ and $svn\_ra\_svn\_readbuf\_input$ to receive data. Thus, for each commit operation, we accumulate the data sent in the write functions, which are the total communication from the client to the server. Similarly, we accumulated the data received in the read functions, which are the total communication from the server to the client. SSVN also relies on these four I/O functions, thus we measured its communication overhead similarly.

The experimental results for the commit phase are shown in Tables III, IV and V. We have several observations: First of all, compared to the non-secure SVN, SSVN adds only a small overhead to the total computation (between 3% and 11% in Table III) and the total communication from the client to the server (between 3% and 7% in Table IV). Secondly, SSVN adds more overhead to the communication from the server to the client because in SSVN the client retrieves data from the server to facilitate the computation of skip deltas during commit; in contrast, for non-secure SVN, the client does not need to compute the skip deltas locally and the server only sends back small control messages. This is the main cost we need to pay for offering a secure version of SVN. Although the communication overhead in Table V is higher for SSVN, we note that in the worst case the additional overhead for committing one version in GCC2 is less than 3KB.

	FileZilla	Wireshark	GCC1	GCC2
SSVN (s)	0.427	0.416	0.417	10.776
non-secure SVN (s)	0.389	0.376	0.386	10.502

TABLE III: The average time for committing one version in both SSVN and non-secure SVN (in seconds).

	FileZilla	Wireshark	GCC1	GCC2
SSVN (KB)	4.599	3.458	4.123	6
non-secure SVN (KB)	4.391	3.246	4.017	5.696

TABLE IV: The average communication from the client to the server for committing one version in both SSVN and non-secure SVN.

	FileZilla	Wireshark	GCC1	GCC2
SSVN (KB)	1.559	1.437	1.047	3.244
non-secure SVN (KB)	0.574	0.58	0.574	0.571

TABLE V: The average communication from the server to the client for committing one version in both SSVN and non-secure SVN.

D. Retrieve Phase

For SSVN and non-secure SVN, we evaluated the computation overhead for the retrieve operation by measuring the time needed to run the shell commands "svn update -r i" (for non-secure SVN) and "ssvn update -r i" (for SSVN) to retrieve a version i by updating version $i-1$. The corresponding experimental results are shown in Table VI. We observe that, compared to non-secure SVN, SSVN adds a reasonable overhead: Table VI shows the time needed to retrieve a version in SSVN increases between 6% and 29% compared to non-secure SVN. Note that this additional time is less than 0.3 seconds in the worst case (for GCC2). The additional overhead is caused by checking the validity of the corresponding version, *i.e.* re-computing the retrieve tags for the affected files in this version and comparing them with the retrieve tags sent back by the server. We did not provide evaluation for communication overhead, since there is no additional communication from the client to the server, and the additional communication from the server to the client will only contains retrieve tags of the affected files in this version (we use HMAC-SHA1 to implement retrieve tags, so only 20 bytes are needed for one retrieve tag).

	FileZilla	Wireshark	GCC1	GCC2
secure SVN (s)	0.0535	0.0453	0.0506	5.086
non-secure SVN (s)	0.0416	0.0376	0.0416	4.779

TABLE VI: The average time for retrieving one version in both secure and non-secure SVN (in seconds).

E. Migrating Repositories from Non-Secure SVN to SSVN

Many commercial and non-commercial projects are using SVN for source control management (*e.g.*, FreeBSD [10], GCC, Wireshark, all the open-source projects in Apache Software Foundation [3], etc.). Such projects already have repositories created based on non-secure SVN. To facilitate the migration from non-secure SVN to SSVN, we built **SSVN-Migrate**, a tool that seamlessly converts an existing non-secure SVN repository into a resopitory for SSVN. SSVN-Migrate works as follows: Starting from the first version (*i.e.*, an empty version), each time it calls "svn update" to check out a new version of the data from the non-secure SVN repository (*i.e.*, version number increased by 1), uses "ssvn add" and "ssvn rm" to update the working copy, and then calls "ssvn commit" to commit the changes into the SSVN repository.

We used SSVN-Migrate to migrate FileZilla, Wireshark and GCC to secure SVN. Table VII shows the time needed for migrating all the first 3000 versions of these SVN repositories. We observe that the time needed for migrating the same collection of versions from different SVN repositories does not vary a lot. One possible reason is that the migration time is mainly determined by the repository size, which is approximately linear to the version number.

Note that our SSVN-Migrate tool tries to re-use as much as possible components we have built for SSVN or existing SVN commands. We believe the results can be significantly improved by optimizing the migration process (*e.g.*, work directly with the raw non-secure and secure repositories), using more powerful hardware, or obtaining additional computing resources from public cloud computing services.

	FileZilla	Wireshark	GCC
total time (s)	1,934	1,909	1,719

TABLE VII: The time for migrating the first 3000 versions of the existing SVN repositories to SSVN (in seconds).

VII. Related Work

Remote data checking for archival storage. As an effective technique for ensuring the integrity of data outsourced at an untrusted party, remote data checking (RDC) has been investigated extensively for both the single-server setting ([26], [43], [46], [28], [39], [27]) and the multiple-server setting ([38], [30], [48], [37]). Recent work on RDC focuses on new topics such as proofs of fault tolerance [32], proofs of location [29], [50], [42] and server-side repair [36].

Dynamic remote data checking. Dynamic Provable Data Possession (DPDP) relies on authenticated data structures (e.g., skip lists [40], RSA trees [40], Merkle trees [49], 2-3 trees [52]) to support the full range of dynamic operations. DPDP adopts spot checking for efficiency and is thus vulnerable to small corruption attack. Follow-up work [35], [34] tries to mitigate such an attack by adding robustness. Concurrently with DPDP, Dynamic Proofs of Retrievability (D-PoR) tries to adapt PoR to a dynamic setting. To support D-PoR, recent work either computes and stores the parity of the data at the client side [47], or relies on Oblivious RAM [33].

Remote data checking for version control systems. Anagnostopoulos et al. [24] introduced the notion of persistent authenticated dictionaries, which allow the user to check whether element e was on set S at time t. Erway et al. [40] adopted a two-level authenticated data structure to provide integrity guarantee for version control systems. Specifically, for each file version, a first-level authenticated data structure is used to organize all of its blocks, generating a root for each version. A second-level authenticated data structure is then used to organize all of these roots. The checking complexity is thus $O(log(tn))$, in which t is the total number of versions and n is the total number of blocks in a version. Etemad et al. [41] improved the solution proposed in [40]. They adopt a PDP-like structure [26], rather than an authenticated data structure, to provide integrity guarantee for the roots of the first-level authenticated data structure, thus reducing the checking complexity to $O(1 + log(n))$. Zhang et al. [51] proposed an update tree-based solution. Their scheme adopts a tree structure to organize all the update operations, and thus the checking complexity is logarithmic in the total number of updates, *i.e.*, approximately $O(log(t))$. In RDC–AVCS, we provide the most efficient solution known to date, which relies solely on an efficient RDC scheme to reduce the checking complexity to $O(1)$.

VIII. Conclusion

In this paper, we introduce Auditable Version Control Systems (AVCS), which are delta-based VCS systems designed to function under an adversarial setting. We propose RDC–AVCS, an AVCS scheme for skip delta-based version control systems, which relies on RDC mechanisms to ensure all the versions of a file can be retrieved from the untrusted VCS server over time. Unlike previous solutions which rely on dynamic RDC and are interesting from a theoretical point of view, our RDC–AVCS scheme is the first pragmatic approach for auditing real-world VCS systems. Our security analysis and experimental evaluation show that RDC–AVCS achieves the desired security guarantees at the cost of a modest decrease in performance compared to a regular (non-secure) VCS system.

Acknowledgment

This research was sponsored by the US National Science Foundation grants CAREER 1054754-CNS and 1241976-DUE. The authors would like to thank Ying Chen for her contribution in the early stages of this work.

References

[1] "Amazon simple storage service," http://aws.amazon.com/en/s3/.
[2] "Apache portable runtime," http://apr.apache.org/.
[3] "The apache software foundation," http://www.apache.org/.
[4] "Apache subversion," http://subversion.apache.org/.
[5] "Bitcasa," https://www.bitcasa.com.
[6] "Code-sharing site github turns five and hits 3.5 million users, 6 million repositories," http://thenextweb.com/insider/2013/04/11/code-sharing-site-github-turns-five-and-hits-3-5-million-users-6-million-repositories/.
[7] "Concurrent versions system," http://cvs.nongnu.org.
[8] "Dropbox," https://www.dropbox.com.
[9] "Filezilla," https://filezilla-project.org/.
[10] "Freebsd," http://www.freebsd.org/.
[11] "Gcc," http://gcc.gnu.org/.
[12] "Git," http://git-scm.com.
[13] "Github," https://github.com.
[14] "Google code," http://code.google.com.
[15] "Mercurial," http://mercurial.selenic.com.
[16] "OpenSSL," http://www.openssl.org/.
[17] "Sourceforge," http://sourceforge.net.
[18] "Summary of the amazon ec2, amazon ebs, and amazon rds service event in the eu west region," http://aws.amazon.com/cn/message/2329B7/.
[19] "Summary of the aws service event in the us east region," http://aws.amazon.com/cn/message/67457/.
[20] "Summer of code 2012 ideas," https://github.com/trast/git/wiki/SoC-2012-Ideas.
[21] "Summer of code 2013 ideas," https://github.com/trast/git/wiki/SoC-2013-Ideas.
[22] "What is sourceforge.net [tm]?" http://sourceforge.net/apps/trac/sourceforge/wiki/What%20is%20SourceForge.net.
[23] "Wireshark," http://www.wireshark.org/.
[24] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia, "Persistent authenticated dictionaries and their applications," in *Information Security*. Springer, 2001, pp. 379–393.
[25] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *Proc. of ACM Conference on Computer and Communications Security (CCS '07)*, 2007.

[26] G. Ateniese, R. Burns, R. Curtmola, J. Herring, O. Khan, L. Kissner, Z. Peterson, and D. Song, "Remote data checking using provable data possession," *ACM Trans. Inf. Syst. Secur.*, vol. 14, June 2011.

[27] G. Ateniese, S. Kamara, and J. Katz, "Proofs of storage from homomorphic identification protocols," in *Proc. of 15th Annual International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT '09)*, 2009.

[28] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik, "Scalable and efficient provable data possession," in *Proc. of International ICST Conference on Security and Privacy in Communication Networks (SecureComm '08)*, 2008.

[29] K. Benson, R. Dowsley, and H. Shacham, "Do you know where your cloud files are?" in *Proc. of ACM Cloud Computing Security Workshop (CCSW '11)*, 2011.

[30] K. Bowers, A. Oprea, and A. Juels, "HAIL: A high-availability and integrity layer for cloud storage," in *Proc. of ACM Conference on Computer and Communications Security (CCS '09)*, 2009.

[31] K. D. Bowers, A. Juels, and A. Oprea, "Proofs of retrievability: Theory and implementation," in *Proc. of ACM Cloud Computing Security Workshop (CCSW '09)*, 2009.

[32] K. D. Bowers, M. V. Dijk, A. Juels, A. Oprea, and R. L. Rivest, "How to tell if your cloud files are vulnerable to drive crashes," in *Proc. of ACM Conference on Computer and Communications Security (CCS '11)*, 2011.

[33] D. Cash, A. Kupcu, and D. Wichs, "Dynamic proofs of retrievability via oblivious ram," in *Proc. of EUROCRYPT '13*, 2013.

[34] B. Chen and R. Curtmola, "Poster: Robust dynamic remote data checking for public clouds," in *Proc. of ACM Conference on Computer and Communications Security (CCS '12)*, 2012.

[35] B. Chen and R. Curtmola, "Robust dynamic provable data possession," in *Proc. of International Workshop on Security and Privacy in Cloud Computing (ICDCS-SPCC '12)*, 2012.

[36] B. Chen and R. Curtmola, "Towards self-repairing replication-based storage systems using untrusted clouds," in *Proc. of ACM Conference on Data and Application Security and Privacy (CODASPY '13)*, 2013.

[37] B. Chen, R. Curtmola, G. Ateniese, and R. Burns, "Remote data checking for network coding-based distributed storage systems," in *Proc. of ACM Cloud Computing Security Workshop (CCSW '10)*, 2010.

[38] R. Curtmola, O. Khan, R. Burns, and G. Ateniese, "MR-PDP: Multiple-replica provable data possession," in *Proc. of International Conference on Distributed Computing Systems (ICDCS '08)*, 2008.

[39] Y. Dodis, S. Vadhan, and D. Wichs, "Proofs of retrievability via hardness amplification," in *Proc. of 6th IACR Theory of Cryptography Conference (TCC '09)*, 2009.

[40] C. Erway, A. Kupcu, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," in *Proc. of ACM Conference on Computer and Communications Security (CCS '09)*, 2009.

[41] M. Etemad and A. Kupcu, "Transparent, distributed, and replicated dynamic provable data possession," in *Proc. of 11th International Conference on Applied Cryptography and Network Security (ACNS '13)*, 2013.

[42] M. Gondree and Z. N. J. Peterson, "Geolocation of data in the cloud," in *Proc. of ACM Conference on Data and Application Security and Privacy (CODASPY '13)*, 2013.

[43] A. Juels and B. S. Kaliski, "PORs: Proofs of retrievability for large files," in *Proc. of ACM Conference on Computer and Communications Security (CCS '07)*, 2007.

[44] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication," internet RFC 2104, February 1997.

[45] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," *ACM Transactions on Computer Systems (TOCS)*, vol. 29, no. 4, p. 12, 2011.

[46] H. Shacham and B. Waters, "Compact proofs of retrievability," in *Proc. of Annual International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT '08)*, 2008.

[47] E. Stefanov, M. van Dijk, A. Oprea, and A. Juels, "Iris: A scalable cloud file system with efficient integrity checks," in *Proc. of Annual Computer Security Applications Conference (ACSAC '12)*, 2012.

[48] C. Wang, Q. Wang, K. Ren, and W. Lou, "Ensuring data storage security in cloud computing," in *Proc. of IEEE International Workshop on Quality of Service (IWQoS '09)*, 2009.

[49] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li, "Enabling public auditability and data dynamics for storage security in cloud computing," *IEEE Trans. on Parallel and Distributed Syst.*, vol. 22, no. 5, May 2011.

[50] G. J. Watson, R. Safavi-Naini, M. Alimomeni, M. E. Locasto, and S. Narayan, "LoSt: location based storage," in *Proc. of ACM Cloud Computing Security Workshop (CCSW '12)*, 2012.

[51] Y. Zhang and M. Blanton, "Efficient dynamic provable possession of remote data via balanced update trees," in *Proc. of 8th ACM Symposium on Information, Computer and Communications Securit (ASIACCS '13)*, 2013.

[52] Q. Zheng and S. Xu, "Fair and dynamic proofs of retrievability," in *Proc. of ACM Conference on Data and Application Security and Privacy (CODASPY '11)*, 2011.

Appendix

A. The Cost for Retrieving an Arbitrary Version in a Skip Delta-based Version Control System

Theorem A.1. *In a skip delta-based version control system, the cost for retrieving an arbitrary version t is bounded by $O(log(t))$.*

Proof: (sketch) According to Figure 1(b), we can always re-compute version $\mathtt{F}_t$ by starting from the initial version $\mathtt{F}_0$, and applying all the corresponding skip deltas up to δ_t. Let l be the total number of skip deltas used to re-construct $\mathtt{F}_t$. Since the skip delta of an arbitrary version is the delta of this version against its skip version, l is thus equal to the total number of skip versions from $\mathtt{F}_0$ up to $\mathtt{F}_t$. According to the rule of determining a version's skip version in Sec. II-A, we can infer that l is actually the total number of bits with value "1" in t's binary format, which is at most $log(t)$. In other words, based on $\mathtt{F}_0$, we need to go through at most $log(t)$ skip deltas to re-compute $\mathtt{F}_t$. Thus, the cost for retrieving an arbitrary version t is bounded by $O(log(t))$. ■

B. The Client Storage for the Inefficient AVCS System

Theorem A.2. *The client storage for the inefficient* AVCS *system is $O(t)$, in which t is the total number of versions in a repository.*

Proof: (sketch) Let $f(t)$ be the total number of versions needed to be stored in the client to facilitate the computation of skip deltas in the inefficient AVCS system. Let $i \leftarrow j$ denote that version i is version j's skip version; similarly $i \rightarrow j$ denotes version j is version i's skip version. Let $b_0 \dots b_i \dots b_{t-1}$ be the binary representation of a version number t, in which b_i is either "0" or "1", *e.g.*, 00 is version number 0's binary representation.

- For $t = 4$, according to the rule of determining the skip version, we have: $01 \rightarrow 00 \leftarrow 10 \leftarrow 11$.
  We can see that by only storing version 0, the client can always compute all the skip deltas locally: The client can compute locally the skip deltas for versions 1 and 2, since the skip version for both of these is version 0; The client can also compute locally the skip delta for version 3, since version 2 (which is the skip

version for version 3), is version 3's immediate previous version. In other words, $f(4) = 1 = 2^0 = 2^{log4-2}$.

- For $t = 8$, we can divide all the 8 versions into 2 groups:
  Group 1, in which the first bit is 0: 001 $\rightarrow$ 000 $\leftarrow$ 010 $\leftarrow$ 011;
  Group 2, in which the first bit is 1: 101 $\rightarrow$ 100 $\leftarrow$ 110 $\leftarrow$ 111.
  We can see that, without considering the first bit, each of the two groups is equivalent to the case of $t = 4$, thus, we can infer that $f(8)$ should be twice compared to $f(4)$: $f(8) = 2 * f(4) = 2 = 2^1 = 2^{log8-2}$. Similarly, for the general case, we have: $f(t) = 2 * (f(\frac{t}{2}))$, by which we can further compute that $f(t) = 2^{log(t)-2} = \frac{t}{4}$.

■