CHAPTER

17 AUDITABLE VERSION CONTROL SYSTEMS IN UNTRUSTED PUBLIC CLOUDS

Bo Chen[*], **Reza Curtmola**[†], **Jun Dai**[‡,§]

[*]*Department of Computer Science, University of Memphis, Memphis, TN, USA* [†]*Department of Computer Science, New Jersey Institute of Technology, Newark, NJ, USA* [‡]*Southwestern University of Finance and Economics, Chengdu, Sichuan, China* [§]*Rutgers University–Newark, Newark, NJ, USA*

17.1 MOTIVATION AND CONTRIBUTIONS

Software development process usually relies on version control systems (VCS) to automate the management of source code, documentation, and configuration files. A version control system can provide several useful features to software developers. This includes retrieving an arbitrary previous version of the source code in order to locate and fix bugs, rolling back to an early version when the working version is corrupted, or allowing team development in which multiple software developers can work simultaneously on updates. In fact, a version control system is indispensable for managing large software projects. Known version control systems which are extensively used nowadays include CVS [3], Subversion [14], Git [8], and Mercurial [11], etc.

A version control system usually records all changes of the data in a data store, called repository, by which an arbitrary version of the data can be retrieved at any time in the future. Oftentimes, repositories are massive in size and difficult to be stored and managed in local machines. For example, GitHub [9] hosted over 6 million repositories, SourceForge over 324,000 projects [13], and Google Code over 250,000 projects [10]. To reduce the cost of both storing and managing repositories, data owners could turn to public clouds providers. As real-world examples, file hosting service providers like Bitcasa [2] and Dropbox [4] that offer version control functionality to their stored data, rely on a popular public cloud provider Amazon S3 [12] for storage services.

The public cloud providers can offer proficient and cheap storage services. However, they are not necessarily trusted due to various reasons. First, they are vulnerable to various attacks from outside or even inside. Second, they usually rely on complex distributed systems, which are likely vulnerable to various failures caused by hardware, software, or even administrative faults [31]. Additionally, unexpected accidental events may lead to failures of cloud services, e.g., power outage [16,17].

Remote Data Integrity Checking (RDIC) [18,19,30] can be used to address the concerns about the untrusted nature of the public cloud providers that host the VCS repositories. RDIC is a mechanism that has been recently developed to check the integrity of data stored at untrusted, third-party storage service providers. Briefly, RDIC allows a client who initially stores a file at a third-party provider (e.g., a cloud provider) to check later if the provider continues to store the original file in its entirety. This check can be done periodically, depending on the client's needs.

Software Architecture for Big Data and the Cloud. DOI: 10.1016/B978-0-12-805467-3.00017-X

By leveraging RDIC, we are able to ensure all the versions of a file are retrievable from the untrusted VCS server over time, so that the version control system can function properly even in the untrusted public clouds. We introduce AVCS, a novel Auditable Version Control System framework that allows the client to periodically verify the VCS repositories and thus obtain the retrievability guarantee. To reduce storage overhead, modern version control systems adopt "delta encoding" to store versions in a repository, in which only the first is stored in its entirety, and each subsequent file version is stored as the difference from its immediate previous version. These differences are recorded in discrete files called "deltas." Particularly, when each version is stored as the difference from another previous version (i.e., not necessarily the immediate previous version), it will turn to a special "skip delta." Such a skip delta-based encoding can further optimize the combine cost of storage and retrieval of an arbitrary file version. This chapter therefore instantiates the AVCS framework for skip delta-based version control systems. Although portions of this chapter are based on previously published work [24], the chapter contains significantly revised material.

Contributions. The contributions of this chapter are summarized in the following:

- We offer a technical overview of both delta-based and skip delta-based version control systems, which have been designed to work under a benign setting. We make the important observation that the only meaningful operation in real-world modern version control systems is append. Unlike previous approaches that rely on dynamic RDIC and are interesting from a theoretical point of view, ours is the first to take a pragmatic approach for auditing the real-world version control systems.
- We introduce the definition of Auditable Version Control Systems (AVCS), and instantiate the AVCS construction for skip delta-based version control systems. Our AVCS construction relies on RDIC mechanisms to ensure all the versions of a file can be retrieved from the untrusted version control server over time, and is able to provide the following features: (i) In addition to the regular functionality of an unsecure version control system, it offers the data owner the ability to check the integrity of all file versions in the VCS repository. (ii) The cost of performing integrity checking on all the file versions is asymptotically the same with the cost of checking one file version. (iii) It can allow the data owner to check the correctness of a file version retrieved from the VCS repository. (iv) It only requires the same amount of storage on the client side like a regular (unsecure) version control system.
- We summarize all the RDIC approaches for version control systems, and theoretically compare them in term of performance.

Organization. In the remainder of this chapter, we introduce our background knowledge in Section 17.2 as well as our system and adversarial model in Section 17.3. In Section 17.4, we introduce the definition of the AVCS, and present our construction of the AVCS for skip delta-based version control systems. We discuss a few issues faced by our AVCS construction in Section 17.5 and summarize the existing RDIC approaches for version control systems in Section 17.6. In Section 17.7, we provide a theoretic comparison among all the RDIC approaches for version control systems; we also perform experimental evaluations on our approach.

17.2 BACKGROUND KNOWLEDGE

17.2.1 DATA ORGANIZATION IN VERSION CONTROL SYSTEMS

In software development, version control systems have been used broadly in managing source code, documentation, and configuration files. Typically, the VCS clients interact with a VCS server and the VCS server stores all the changes to the data in a main repository, such that an arbitrary 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 (i.e., the version of the data that was last checked out by the client from the main VCS repository), the changes made by the client to the working copy, and some metadata.

A version control system provides several useful features that can allow the users to track and control changes made to the data over time. Such features include operations like commit, update, branch, revert, merge, and log. In practice, the most commonly used operations are *commit* and *retrieve*. Commit refers to the process of submitting the latest changes of 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.

Version control systems using delta encoding. A version control system keeps track of all the changes made to the data over time. In this way, any version of the data can be retrieved if necessary. A key issue here is how to store and organize all those changes in the VCS repository. A straightforward approach could be simply storing each individual data version upon a commit. This simple approach was adopted by CVS [3] when storing binary files. A significant disadvantage of this approach is the large bandwidth and storage space it requires for committing a file version, especially when the file size is large.

By observing that there is always a large amount of duplicate content between each file version and its subsequent file version, modern version control systems adopt "delta encoding" to reduce the storage space being needed. Using delta decoding, a version control system only stores the first version of a file in its entirety, and for each subsequent version of the file, the VCS only stores the difference between this file version and its immediate previous version. All the differences are recorded in discrete files called "deltas."

Therefore, if there are t file versions in total, the VCS server will store them as the initial file and $t-1$ deltas (see Fig. 17.1). Popular version control systems that use variants of delta encoding include CVS [3], SVN [14], and Git [8]. The delta encoding can significantly reduce the storage space required to store all the file versions. However, it suffers from an expensive retrieval: To retrieve the file version t, the VCS server needs to start from the initial file version and apply all the subsequent deltas up to version t, incurring a cost linear in t (see Fig. 17.1). Considering that source code repositories may have a large number of file versions (e.g., GCC has more than 200,000 versions in its repository [7]), retrieving an arbitrary file version may bring a significant burden to the VCS server.

Version control systems using skip delta encoding. As a special type of delta encoding, the skip delta encoding is designed to further optimize the retrieval cost. Apache Subversion (SVN [14]) is a popular version control system that adopts this type of delta encoding. In skip delta encoding, each new file version is still stored as the difference between it and a previous file version. The main difference between the skip delta encoding and the delta encoding is that this previous file version is not necessarily the immediate previous version. Instead, it can be an arbitrary file version from the initial file version up to the immediate previous file version. This ensures that retrieval of an arbitrary file version t requires significantly less than t applications of deltas by the version control server. In this case, the

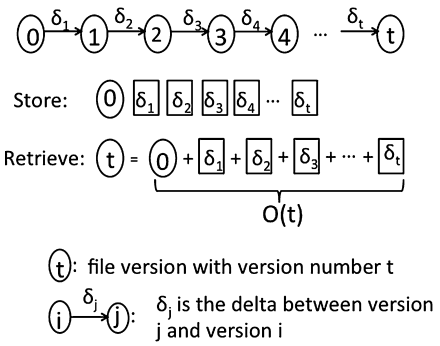

FIGURE 17.1

A delta-based version control system

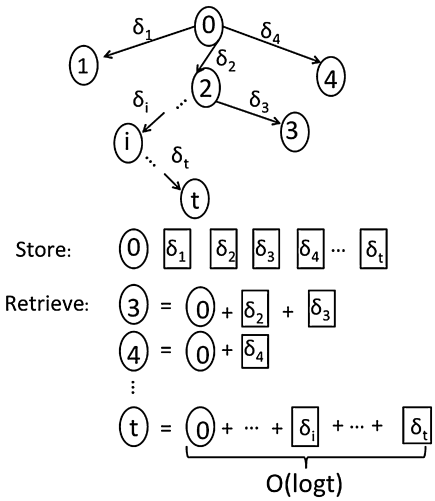

FIGURE 17.2

A skip delta-based version control system

difference is called a "skip delta" and the old version against which a new version is encoded is called a "skip version."

In particular, if we select the skip version based on the following rule, the complexity for retrieving an arbitrary file version t will be O(log(t)), i.e., we only need to apply at most log(t) skip deltas in order to recompute the desired file version t starting from the initial file version. Let version j be the skip version of version i. The rule for selecting the skip version j is: Based on the binary representation of i, we change the rightmost bit that has value "1" into a bit with value "0". For example, in Fig. 17.2, version 3's skip version is version 2. This is because the binary representation of 3 is 011, and by changing the rightmost "1" bit into a "0" bit, we can obtain 2.

By using the skip delta encoding, the cost for recovering an arbitrary file version will be logarithmic in the total number of versions. For example, in Fig. 17.2, to reconstruct version 3, we can start from version 0 and apply δ_2 and δ_3; to reconstruct version 4, we can 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, we can start from version 0 and apply δ_{16}, δ_{24}, and δ_{25}. We prove that the cost for retrieving an arbitrary file version t is bounded by $O(\log(t))$ [24].

17.2.2 REMOTE DATA INTEGRITY CHECKING (RDIC)

Remote Data Integrity Checking (RDIC) is framework that allows a data owner to verify the integrity of the data outsourced to an untrusted third party. RDIC usually relies on two key steps: (i) Data pre-processing. The data to be outsourced are pre-processed by the data owner, generating metadata (e.g., integrity tags) which can be used to verify whether the data have been tampered with over time; (ii) Periodical checking. After the data have been outsourced, a verifier (e.g., the data owner or a third-party verifier) will periodically issue a challenge request to ask the storage server to prove possession of the outsourced data. The existing PDP [18,19] and PoR [30,20] are two popular RDIC designs, in which the checking step is highly optimized by utilizing a novel "spot-checking" technique. Rather than simply verify the entire outsourced data, which may be prohibitively expensive, the spot-checking technique checks a random subset of the entire outsourced data. Prior work [18,19] shows that if an adversary corrupts a certain amount of the data, the spot-checking technique can detect such a corruption with high probability by checking a few randomly selected blocks.

RDIC has been recently extended to a dynamic setting [22,25,27,33] and a distributed setting [21, 23,26,29].

17.3 SYSTEM AND ADVERSARIAL MODEL

System model. We consider a version control system in which one or more VCS clients store data at a VCS server. The server maintains the main repository, in which all the versions of the data are stored. Each VCS client runs client software. We use the term *client* to refer to the VCS client software and *server* to refer to the server software. Each VCS client maintains a local repository, which stores the working copy, the changes made by the client to the working copy, and the metadata.

From a client's point of view, the interface exposed by the VCS server includes two main operations: *commit* and *retrieve*. Although a VCS system usually provides additional operations including branch, merge, log, etc., we only focus on the most common operations commit and retrieve. We also introduce an additional operation, *check*. This operation allows the client to check if the server possesses all the file versions.

Adversarial model. We consider an adversarial model in which all the VCS clients are trusted and the VCS server may misbehave. This captures real-world scenarios in which the software engineers from a company are collaborating to develop software, and each of them will honestly use the VCS client. However, the VCS server may be hosted in a public cloud provider to reduce operational cost. Due to the untrusted nature of public cloud providers, the VCS server may not function properly.

We consider a rational and economically motivated adversary who may delete the data that were rarely used. The attacks are meaningful only when the adversary can obtain a significant profit. We do

not consider attacks in which the server simply corrupts a small portion of the repository (e.g., 1 byte), since such attacks will not bring a significant benefit.

Assumptions. Our design relies on two main assumptions: (i) All the communications are protected (e.g., by SSL/TLS), and the adversary is not able to learn anything from eavesdropping the communications; (ii) The server should respond to any valid requests from the clients. Otherwise, the clients can simply terminate the contract with the service provider.

17.4 AUDITABLE VERSION CONTROL SYSTEMS

A version control system designed for a benign setting cannot work correctly when the VCS server is untrusted. This is usually the case when the server is hosted in the public clouds. We thus introduce a new framework, Auditable Version Control System (AVCS), which can ensure that a version control system is able to function properly even when the VCS server is untrusted and misbehaves.

In this section, we first introduce our definition of the AVCS. By leveraging remote data integrity checking mechanisms, we then instantiate the construction of AVCS for skip delta-based version control systems.

17.4.1 DEFINITION OF AVCS

The AVCS relies on seven polynomial-time algorithms: *KeyGen*, *ComputeDelta*, *GenMetadata*, *GenProof*, *CheckProof*, *GenRetrieveVersionAndProof*, and *CheckRetrieveProof*.

KeyGen is a key generation algorithm run by the client to initiate the entire version control system.

ComputeDelta is a delta generation algorithm run by the client to compute the corresponding delta when committing a new file version.

GenMetadata is a metadata generation algorithm run by the client to generate the verification metadata for a new file version before committing it.

GenProof is an algorithm run by the server to generate a proof of data possession.

CheckProof is an algorithm run by the client to verify the proof of data possession.

GenRetrieveVersionAndProof is an algorithm run by the server to generate the correctness proof of a retrieved file version.

CheckRetrieveProof is an algorithm run by the client to verify the correctness of the retrieved file version.

Utilizing the aforementioned algorithms, we build an AVCS divided into 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 store at the server.

Challenge. Periodically, the verifier (the client or a third-party verifier) challenges the server to obtain a proof that the server continues to store all the file versions stored by the client. The server uses *GenProof* to compute a proof of data possession and sends back the proof. The client then uses *CheckProof* to validate the received proof.

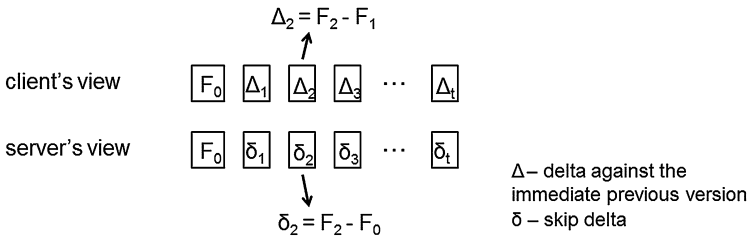

FIGURE 17.3

Inconsistency of the client's and the server's view

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

Note that this definition encompasses version control systems that use delta encoding, which also include skip delta-based version control systems.

17.4.2 AN AVCS CONSTRUCTION

Whereas our AVCS definition targets version control systems that use delta encoding in general, in the construction we focus on version control systems that use skip delta encoding, because: First, they are optimized for both storage and retrieval; Second, they are arguably more challenging to secure than version control systems that use delta encoding, due to the additional complexity of computing the skip deltas.

Challenges. To instantiate the AVCS construction for skip delta-based version control systems, we need to address several challenges. These challenges stem from the adversarial nature of the VCS server that is hosted in the untrusted clouds and from the format of a skip delta-based VCS repository being optimized to minimize the server's storage and workload during the Retrieve phase.

The first challenge comes from the gap between the server's and the client's view of the repository. In a general-purpose RDIC protocol (Section 17.2.2), 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. Additionally, the server stores and uses the data and metadata to answer the verifier's challenges by computing a proof that convinces the verifier that the server continues to store the exact data outsourced by the data owner.

However, in a skip delta-based VCS, there is a gap between the two views (Fig. 17.3), which makes skip delta-based VCS systems more difficult to audit: Although both the 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, i.e., the difference between version t and $t - 1$. However, this delta is different from the skip deltas that are stored by the server: a δ_t file stored by the server is the difference between version t and its "skip version", which is not necessarily the version immediately previous to

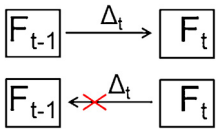

FIGURE 17.4

Delta-encoding is not reversible

version t. 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 the RDIC protocol.

The second challenge is due to the fact that 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 the delta encoding is not a reversible operation (Fig. 17.4). Specifically, if $\delta_{t-1\to t}$ is the difference between versions $t-1$ and t (i.e., $F_t = F_{t-1} + \delta_{t-1\to t}$), this does not imply that version $t-1$ can be obtained based on version t and $\delta_{t-1\to t}$. The reason comes from the method used by delta encoding to encode update operations between versions, such as insert, update, and delete. If a delete operation was executed on version $t-1$ to obtain version t, then $\delta_{t-1\to t}$ encodes only the position of the deleted portion from version $t-1$, so that given version $t-1$ and $\delta_{t-1\to t}$, one can obtain version t. However, $\delta_{t-1\to t}$ does not encode the actual data that has been deleted. Thus, version $t-1$ cannot be obtained based on version t and $\delta_{t-1\to t}$.

High-level idea. We present an AVCS construction that can resolve the aforementioned challenges. The idea behind our construction is that we allow the client and the untrusted server to collaborate to compute the skip deltas. By sacrificing an acceptable amount of communication, we successfully eliminate the need of storing a large number of previous versions in the client (for the purpose of computing skip deltas), and thus are able to conform to the notion of storage outsourcing.

17.4.2.1 Construction details

The version control repository stores t file versions, $F_0, F_1, \ldots, F_{t-1}$. They are stored in the repository using skip delta encoding as: $F_0, \delta_1, \delta_2, \ldots, \delta_{t-1}$ (i.e., the initial file version and $t-1$ skip delta files). We view the entire information pertaining to the t versions of the file F as a virtual file $\tilde{F}$, which is obtained by concatenating the original file version and all the $t-1$ subsequent delta files: $\tilde{F} = F_0||\delta_1||\delta_2||\ldots||\delta_{t-1}$. We view $\tilde{F}$ as a collection of fixed-size blocks, and each block containing s symbols (a symbol is an element from $GF(p)$, where p is a large prime (at least 80 bits)). This view matches the view of a file in an RDIC scheme: To check the integrity of all the versions of F, it is enough to check the integrity of $\tilde{F}$. Let n denote the number of blocks in $\tilde{F}$. As the client keeps committing new file versions, n will grow accordingly (note that n is maintained by the client).

We use two types of verification tags. To check data possession (in the Challenge phase), we use *challenge tags*, which are computed over the blocks in $\tilde{F}$ to facilitate spot checking technique in RDIC (Section 17.2.2). The challenge tag is computed for each block following the manner of private verification tag construction in [32], which is homomorphically verifiable. To check the integrity of individual file versions (in both the Commit and the Retrieve phases), we use *retrieve tags*, each of which is com-

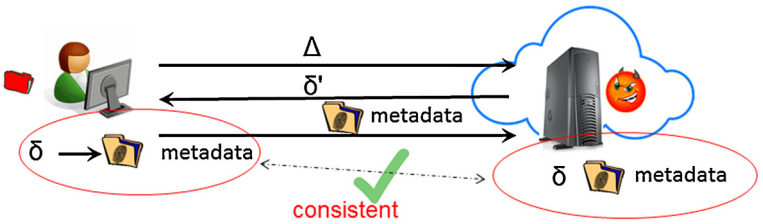

FIGURE 17.5

Overview of the AVCS construction

puted over an entire version of F. The retrieve tag is computed using HMAC [29] over the whole file version and its version number.

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. Under an adversarial setting, to leverage RDIC techniques over the VCS repository, the skip delta files must be accompanied by challenge tags. Since the challenge tags can only be computed by the client, in our construction, we require the client to compute the skip delta, generate the challenge tags over it, and send both the skip delta and the tags to the server. An overview of our AVCS construction is shown in Fig. 17.5.

In the following, we elaborate the design of our AVCS construction for each phase: setup, commit, challenge, and retrieve.

Setup. The client selects two private keys K_1 and K_2 uniformly at random from a large domain (this is determined by the security parameters). It also initializes n as 0.

Commit. As shown in Fig. 17.6, when committing a new version F_i, where $i > 0$, the client must compute the skip delta (δ_{skip}) for F_i. The δ_{skip} must be computed against a certain previous version of the file, called the "skip version" (refer to Section 17.2.1). Let $skip(i)$ be the skip version of version i. Note that the client also has in its local store a copy of F_{i-1}, which is the working copy.

The client will check if $skip(i)$ is version $i-1$. If it is true, the client can directly compute δ_{skip} such that $F_i = F_{i-1} + \delta_{skip}$. Otherwise, the client computes δ_{skip} by interacting with the VCS server using the following steps (see Fig. 17.6):

1. The client computes the difference between the new version and its immediate previous version, i.e., computes δ such that $F_i = F_{i-1} + \delta$. The client then sends δ to the server.
2. The server recomputes F_{i-1} based on the data in the repository and then computes $F_i = F_{i-1} + \delta$. The server then recomputes $F_{skip(i)}$ (the skip version for F_i) based on the data in the repository and computes the difference between F_i and $F_{skip(i)}$, i.e., it computes $\delta_{reverse}$ such that $F_{skip(i)} = F_i + \delta_{reverse}$. The server sends $\delta_{reverse}$ to the client, together with the retrieve tag for $F_{skip(i)}$.
3. The client computes the skip version, $F_{skip(i)} = F_i + \delta_{reverse}$, and checks the validity of $F_{skip(i)}$ 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 $F_i = F_{skip(i)} + \delta_{skip}$.

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

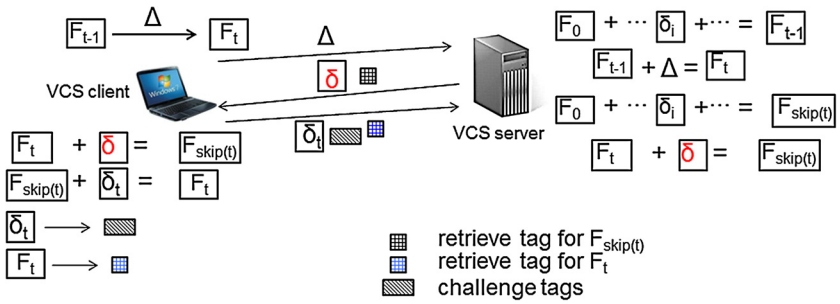

FIGURE 17.6

Commit phase of our AVCS construction

$\delta_{reverse}$ and then compute F_{16}, which is the skip version for F_{20}, as $F_{16} = F_{20} + \delta_{reverse}$. Only then can the client compute δ_{skip} such that $F_{20} = F_{16} + \delta_{skip}$.

After having computed δ_{skip}, the client views δ_{skip} as a collection of blocks, and computes a set of challenge tags using key K_1. The client also computes a retrieve tag R_i for F_i, using key K_2. The set of challenge tags and the retrieve tag R_i will be sent to the VCS server. The client then increases n by x, where x is the number of blocks in δ_{skip}. Note that if the file version being committed is the initial file version F_0, the client directly computes a set of challenge tags and a retrieve tag over F_0.

Challenge. We leverage spot checking technique (see Section 17.2.2) to check the remotely stored CVS repository. Periodically, the client challenges the server to prove data possession of a random subset of the blocks in $\tilde{F}$. The server computes a proof of data possession by using the challenge blocks and their corresponding challenge tags. As the challenge tags are homomorphically verifiable, the server can further reduce the size of the proof by aggregating the challenge blocks and the corresponding challenge tags. The client further checks the proof using key K_1. This spot checking mechanism is quite efficient. For example, when the server corrupts 1% of the repository (i.e., 1% of $\tilde{F}$), 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) [18].

Retrieve. The Retrieve phase is activated when the client wants to replace the working copy with an older or a newer version F_j. The client sends a retrieval request to the server. The server generates the delta (i.e., δ) of F_j against the client's working copy, together with F_j's retrieve tag R_j. Both δ and R_j are returned to the client. The client then computes F_j by applying δ over its working copy, and checks the validity of F_j by using R_j and the key K_2.

17.5 DISCUSSION

Broad applications of the AVCS. One significant advantage of skip delta encoding is that it can be used to handle any formats of data, including source code, audio, image, video, etc. As the AVCS is based on skip delta encoding, it can securely handle any types of data that need to be maintained in a version control manner and need to be stored in an untrusted third party like a public cloud provider.

SVN vs Git. Although Git has been recently gaining popularity compared to SVN, providing guarantees to SVN repositories is still very valuable because there are still plenty of projects and organization that still use SVN. The techniques described in this chapter could be adapted to be applied to Git repositories; we anticipate that devising similar techniques for Git will present fewer challenges, since Git clients clone the entire repository (i.e., all revisions), unlike in SVN where the client usually has only one revision (out of many revisions).

17.6 OTHER RDIC APPROACHES FOR VERSION CONTROL SYSTEMS

DPDP. Erway et al. [27] used 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 the blocks, generating a root for each version. A second-level authenticated data structure is then used to organize all these roots. As the total number of leaves in this two-level authenticated data structure is $t \cdot n$, and verifying a block in the tree-like authenticated data structure usually has a logarithmic complexity, the checking complexity of DPDP will be $O(\log(t \cdot n))$, in which t is the total number of versions and n is the total number of blocks in a version.

DR-DPDP. Etemad et al. [28] proposed DR-DPDP to improve the performance of DPDP. In this approach, they use a first level authenticated data structure to organize all the data blocks of a file version, generating the corresponding root. They then adopt a PDP-like structure [18], rather than an authenticated data structure, to provide integrity guarantee for the roots of the first-level authenticated data structure. Since checking the roots in PDP-like structure can be achieved in constant time, the checking complexity of DR-DPDP can be significantly reduced to $O(1 + \log(n))$.

Other work. Zhang et al. [34] proposed an update tree-based approach. 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 $O(\log(t))$.

17.7 EVALUATION

17.7.1 THEORETICAL EVALUATION

Theoretical analysis on our AVCS construction. During the Commit phase, the client interacts with the version control server to compute skip deltas. To retrieve any file version from the repository, the server only needs to go through at most $\log(t)$ skip deltas. Therefore, the computation complexity in the server is $O(\log(t))$. The client has to compute the skip version as well as the skip delta, and generate the metadata, which results in a complexity linear to the size of the file version. The communication during the commit phase includes two deltas and a set of challenge tags for a skip delta.

During the Challenge phase, our AVCS uses an efficient spot checking technique: periodically, the client challenges the server, requiring the server to prove data possession of a random subset of data blocks in the entire repository; the server computes a proof by aggregating the selected blocks as well as the corresponding challenge tags. Therefore, both the computation (client and server) and the communication complexities are $O(1)$. This is a major advantage of the AVCS compared to all the

Table 17.1 Comparison of different RDIC approaches for version control systems (t is the number of versions in the repository and n is the number of blocks in a version)

	DPDP [27]	DR-DPDP [28]	AVCS
Communication (Commit phase)	$O(n + \log t)$	$O(n + 1)$	$O(n + 1)$
Server computation (Commit phase)	$O(n + \log t)$	$O(n)$	$O(n \log t)$
Client computation (Commit phase)	$O(n + \log t)$	$O(n + 1)$	$O(n + 1)$
Communication (Challenge phase)	$O(\log n + \log t)$	$O(1 + \log n)$	$O(1)$
Computation (server + client) (Challenge phase)	$O(\log n + \log t)$	$O(1 + \log n)$	$O(1)$
Communication (Retrieve phase)	$O(n + \log t)$	$O(n + 1)$	$O(n + 1)$
Server computation (Retrieve phase)	$O(tn + \log t)$	$O(tn + 1)$	$O(n \log t + 1)$
Client computation (Retrieve phase)	$O(n + \log t)$	$O(n)$	$O(n)$
Client storage	$O(n)$	$O(n)$	$O(n)$
Server storage	$O(nt)$	$O(nt)$	$O(nt)$

previous approaches, in which the checking complexity is proportional to either the repository size or the version size.

During Retrieve phase, the server needs to go through at most $\log(t)$ skip deltas to compute a target file version. Therefore, the server computation complexity is $O(\log(t))$. The client stores locally the working copy which requires $O(n)$ space.

Comparison among different secure version control approaches. We show a performance comparison among our AVCS construction, DPDP [27], and DR-DPDP [28] in Table 17.1. Compared to both DPDP and DR-DPDP, our AVCS has constant complexity in the Challenge phase in terms of both computation and communication. In the Retrieve phase, the AVCS is significantly more efficient than the other approaches in terms of server computation. However, the server computation in the Commit phase of the AVCS is slightly larger than the other approaches due to the overhead for computing the skip deltas. We believe this is not a significant drawback, because: (i) Compared to the other approaches, the AVCS only has an additional logarithmic factor, which is usually small, i.e., for $t = 1{,}000{,}000$, $\log(t)$ is 20; (ii) The server is hosted in the public clouds, which are usually capable of handling computation intensive tasks.

17.7.2 EXPERIMENTAL EVALUATION

To understand the impact of AVCS on the real-world VCS systems, we implemented AVCS using a popular open-source version control system, Apache Subversion (SVN) [14] version 1.7.8. As we know, many projects are using SVN for source control management. This includes FreeBSD [6], GCC [7], Wireshark [15], as well as all the open-source projects in Apache Software Foundation [1], etc.

Implementation. We modified the source code of both the client and the server of SVN. For the client, we mainly modified five SVN commands: *svn add*, *svn rm*, *svn commit*, *svn co*, and *svn update*. For the server, we modified the stand-alone server "*svnserve*." The implementation details as well as implementation issues can be found in our conference paper [24].

Experimental results: We evaluated both the computation and the communication overhead during the Commit phase and the computation overhead during the Retrieve phase, for both SSVN and SVN. We selected three representative public SVN repositories for our experimental evaluation: FileZilla [5],

Wireshark [15], and GCC [7]. The Challenge phase has been shown to be very efficient when using spot-checking technique [19], so we do not include it in this evaluation. The detailed experimental results can be found in our conference paper [24], which shows that AVCS only incurs a modest decrease in performance to the original SVN system.

17.8 CONCLUSION

In this chapter, we introduce Auditable Version Control System (AVCS), a delta-based version control system designed to function properly in untrusted public clouds. By leveraging remote data integrity checking mechanisms, we instantiate our AVCS for skip delta-based version control systems. Unlike previous approaches that rely on dynamic RDIC and are interesting from a theoretical point of view, ours is the first pragmatic approach for auditing real-world VCS systems. We also summarize all the other RDIC approaches for version control systems, and compare them theoretically.

REFERENCES

[1] APACHE – The Apache Software Foundation. Available at apache.org.
[2] Bitcasa. Available at bitcasa.com.
[3] CVS – Concurrent Versions System. Available at cvs.nongnu.org.
[4] Dropbox. Available at dropbox.com.
[5] Filezilla. Available at filezilla-project.org.
[6] FreeBSD. Available at freebsd.org.
[7] GCC. Available at gcc.gnu.org/.
[8] Git. Available at git-scm.com.
[9] GitHub. Available at github.com.
[10] Google code. Available at code.google.com.
[11] Mercurial. Available at mercurial.selenic.com.
[12] S3. Available at aws.amazon.com/en/s3.
[13] Sourceforge. Available at sourceforge.net.
[14] SVN – Apache subversion. Available at subversion.apache.org.
[15] Wireshark. Available at wireshark.org.
[16] AWS MESSAGE 1 – Summary of the Amazon EC2, Amazon EBS, and Amazon RDS service event in the EU West region. Available at aws.amazon.com/cn/message/2329B7.
[17] AWS MESSAGE 2 – Summary of the AWS service event in the US East region. Available at aws.amazon.com/cn/message/67457.
[18] G. Ateniese, R. Burns, R. Curtmola, J. Herring, O. Khan, L. Kissner, Z. Peterson, D. Song, Remote data checking using provable data possession, ACM Trans. Inf. Syst. Secur. 14 (June 2011).
[19] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, D. Song, Provable data possession at untrusted stores, in: Proc. of ACM Conference on Computer and Communications Security (CCS'07), 2007.
[20] K.D. Bowers, A. Juels, A. Oprea, Proofs of retrievability: theory and implementation, in: Proc. of ACM Cloud Computing Security Workshop (CCSW'09), 2009.
[21] K. Bowers, A. Oprea, 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.
[22] D. Cash, A. Kupcu, D. Wichs, Dynamic proofs of retrievability via oblivious RAM, in: Proc. of EUROCRYPT'13, 2013.
[23] Bo Chen, Anil Kumar Ammula, Reza Curtmola, Towards server-side repair for erasure coding-based distributed storage systems, in: The Fifth ACM Conference on Data and Application Security and Privacy (CODASPY'15), 2015.

[24] B. Chen, R. Curtmola, Auditable version control systems, in: Proc. of the 21st Annual Network and Distributed System Security Symposium (NDSS'14), 2014.
[25] B. Chen, R. Curtmola, Robust dynamic provable data possession, in: Proc. of International Workshop on Security and Privacy in Cloud Computing (ICDCS-SPCC'12), 2012.
[26] Bo Chen, Reza Curtmola, Towards self-repairing replication-based storage systems using untrusted clouds, in: The Third ACM Conference on Data and Application Security and Privacy (CODASPY'13), 2013.
[27] C. Erway, A. Kupcu, C. Papamanthou, R. Tamassia, Dynamic provable data possession, in: Proc. of ACM Conference on Computer and Communications Security (CCS'09), 2009.
[28] M. Etemad, 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.
[29] H. Krawczyk, M. Bellare, R. Canetti, HMAC: keyed-hashing for message authentication, Internet RFC 2104, February 1997.
[30] A. Juels, B.S. Kaliski, PORs: proofs of retrievability for large files, in: Proc. of ACM Conference on Computer and Communications Security (CCS'07), 2007.
[31] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, M. Walfish, Depot: cloud storage with minimal trust, ACM Trans. Comput. Syst. 29 (4) (2011) 12.
[32] H. Shacham, 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.
[33] E. Stefanov, M. van Dijk, A. Oprea, A. Juels, Iris: a scalable cloud file system with efficient integrity checks, in: Proc. of Annual Computer Security Applications Conference (ACSAC'12), 2012.
[34] Y. Zhang, M. Blanton, Efficient dynamic provable possession of remote data via balanced update trees, in: Proc. of 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS'13), 2013.