Confidentiality and Authenticity for Distributed Version Control Systems — A Mercurial Extension

Michael Lass
Paderborn University
33098 Paderborn, Germany
michael.lass@uni-paderborn.de

Dominik Leibenger
CISPA, Saarland University
66123 Saarbrücken, Germany
dominik.leibenger@uni-saarland.de

Christoph Sorge
CISPA, Saarland University
66123 Saarbrücken, Germany
christoph.sorge@uni-saarland.de

***Abstract*—Version Control Systems (VCS) are a valuable tool for software development and document management. Both client/server and distributed (Peer-to-Peer) models exist, with the latter (e.g., Git and Mercurial) becoming increasingly popular. Their distributed nature introduces complications, especially concerning security: it is hard to control the dissemination of contents stored in distributed VCS as they rely on replication of complete repositories to any involved user.**

We overcome this issue by designing and implementing a concept for cryptography-enforced access control which is transparent to the user. Use of field-tested schemes (end-to-end encryption, digital signatures) allows for strong security, while adoption of convergent encryption and content-defined chunking retains storage efficiency. The concept is seamlessly integrated into Mercurial—respecting its distributed storage concept—to ensure practical usability and compatibility to existing deployments.

I. Introduction

Version Control Systems (VCS) have been used for a long time now to manage different versions of files. The *Source Code Control System (SCCS)* [17] came up in 1972: It allows reconstruction of a single file's full version history by storing so-called interleaved deltas instead of replacing the file. Changes can be tagged with metadata (usually timestamp, author name, comment) to provide more complete information.

Over time new systems introduced additional functionality and new concepts. The *Concurrent Versions System (CVS)* [9] tracks the version history of multiple files in a central repository not necessarily located at the user's local workstation. It allows collaboration by coordinating changes made by different users. Users maintain a local *working copy* consisting of a single version of each file from the repository. Repository access and synchronization of changes is performed using the operations *commit* and *checkout*. *Subversion (SVN)* [3] utilizes a delta-based storage structure for tracking version history of entire directory trees, respecting relationships between different files.

If repositories are shared between users, security requirements gain relevance. If, e.g., data is stored in a repository that should only be accessible by some users of the system, strong security requirements apply to the repository server. Access control in existing centralized VCS is, if at all, realized using trivial server-side access control lists (ACLs), totally relying on the server's trustworthiness and integrity. To the best of our knowledge, the sole available work focusing on VCS security is a cryptography-based access control solution for SVN that enforces access rights using end-to-end encryption. [13]

Since *Git* [7] was released in 2005, *distributed VCS* have gained more and more popularity. Mercurial [16] and Git are the most-popular such systems today. In contrast to modern centralized VCS, repositories are stored on users' local workstations again. Collaboration among users is supported by allowing users to synchronize their repositories with others. Revisions can be *pulled* from / *pushed* to remote repositories. There are no limitations concerning the resulting communication paths: Distributed VCS support centralized setups, fully distributed peer-to-peer operation, and hybrid approaches.

The mentioned security concerns are only insufficiently addressed by current distributed VCS: In a peer-to-peer setup, each user has to make sure that revisions are only pushed to other users if they are allowed to access all contained data; communication paths are thereby restricted, as *all* nodes on the paths must be sufficiently trusted. In a centralized setup, on the other hand, all users would have to trust the central server. Requiring such a setup would defeat the purpose of a distributed system. Effective access control can thus only be achieved using cryptographic measures, e.g., end-to-end encryption, which is not supported by any distributed VCS.

We fill this gap with a cryptography-based access control solution for distributed VCS based on [13], achieving confidentiality and authenticity while maintaining storage efficiency:

- We work out how file-level access control can be integrated into the distributed VCS workflow as to allow support for confidential files not intended to be accessed by other legitimate users of the system.
- We present a concept for retrofitting differentiated read and write access rights for files in existing, distributed VCS without loss of storage efficiency. The concept allows legitimate repository users to create confidential files and to manage their access rights over time.
- We achieve authenticity of data and metadata and confidentiality of file contents and file names.
- We transfer our concept into a functional extension for Mercurial that is compatible to conventional repositories, including code hosting platforms like Bitbucket [4].

The paper is structured as follows: Section II specifies precise goals and presents the threat model. Section III gives an in-depth discussion of our concept, followed by the im-

plementation of an extension for Mercurial in Section IV. Security and performance are evaluated in Sections V and VI. After discussing related work in Section VII, we conclude in Section VIII.

II. Goals and Threat Model

Our goal is to enable handling of confidential files in distributed VCS by providing a suitable cryptography-enforced access control mechanism. Any user of a repository shall be able to mark newly created files as confidential, and to manage rights for these files afterwards. We call these users *file owners*. With any new revision, rights can be changed arbitrarily, but they are immutable for a given revision. For any file x marked confidential, we provide security guarantees with respect to each revision r. Let u_o be the user who marked x confidential in a revision r^*. We distinguish six user categories:

1) The owner u_o of the file x.
2) Users u_{rw} with read and write access to file revision r.
3) Users u_r with read access to the file x in revision r.
4) Users $u_{r'}$ who are not allowed to access x in revision r, but in any other revision $r', r' \geq r^*$ ($r' < r$ or $r' > r$).
5) Users u_{na} with access to a repository containing revision r, but without access rights to x in any revision.
6) Others.

Note that the information available to a user of category i is a subset of that available to a user of category $i-1$.

Our security guarantees are as follows:

- Authenticity is guaranteed both for the file name of and access rights to x: If assigned by u_o in revision r or earlier, changes by users other than u_o are detectable.
- Authenticity is guaranteed for file contents: Changes made by users other than u_o or u_{rw} can be detected.
- Confidentiality is guaranteed for file names: Users u_{na} must not get access to file names. We require CCA-secure encryption of file names, i.e., resistance to chosen-ciphertext attacks.
- Confidentiality is guaranteed for file contents: We allow users u_{rw} to choose a trade-off between confidentiality (i.e., CCA-secure encryption) and facilitation of data deduplication. Depending on the trade-off, users $u_{r'}$ must not learn anything about contents or may identify contents they already know. Formally, we require CDPA$_d$-secure encryption (see [13]), i.e., resistance to *chosen different plaintext attacks*: Ciphertexts must not leak any information unless contents with (deduplicable) overlapping sequences of at least d bytes have been encrypted.

Note that we do not aim to provide any guarantees that go beyond individual files, or concerning revisions of files not marked confidential. Further, rights are strictly tied to (and in fact stored with) specific revisions: Users can lose rights in future revisions, but cannot be prevented from reading (or creating successors of) revisions they had access to before.

Moreover, we do not address compromised computer systems, although obviously an attacker who has compromised a system shall not be able to gain more rights than the system's respective user. Integrity of a file owner's computer system is essential. As a consequence, each of the groups 2–6 listed above are considered attackers, differing only in their rights and information available to them. All attackers are assumed to have full read access to their local repository and working copy and to foreign repositories, and full write access to their local repository and working copy. With push/pull, they can synchronize their repository to any other.

III. General Concept

We first describe the general functionality and typical use cases of distributed VCS. Following this, we discuss how access control can be included without loss of compatibility.

A. Prerequisites

We work out the main concepts which are shared by all popular distributed VCS today. Figure 1 illustrates a typical workflow: Alice commits changes in her working copy to her local repository and pushes them to Bob's repository. Bob can check them out. Carol pulls Alice's changes from Bob's repository, performs local changes and pushes them directly to Alice. Usage of a central repository is possible, but optional.

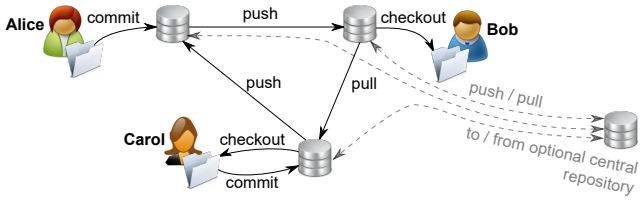

Fig. 1. Typical workflow in distributed VCS

Starting with an initial (empty) revision, all revisions of a project are organized in a *revision graph*. Each commit yields a revision node which is connected to its *base revision*, i.e., the most-recently checked out revision the committed changes are based on. Revisions resulting from a merge of two revisions (with a shared ancestor) have two base revisions.

Revisions are identified by *revision numbers*. To ensure uniqueness despite the system's distributed nature (local repositories might contain only parts of a revision graph), they are computed as *cryptographic hashes* of their revisions' contents, including ancestors. A revision's number thus also guarantees integrity of its version history in absence of hash collisions.

To deal with large revision graphs, VCS avoid multiple storage of identical data. Mercurial represents revisions as deltas to previous ones to store only actual changes [16, Chapter 4], Git deduplicates identical contents and groups similar contents into compressed packfiles [7, Chapter 10.4].

B. Confidentiality Concept

Files marked *confidential* must not be accessed by unauthorized users. While this could be achieved by performing access control at the interface between working copy and repository, i.e., during commit/checkout, this would be insufficient since these operations are executed locally and thus easy to manipulate. Effective access control could be achieved at the

interface between repositories, i.e., during push/pull, preventing transmission of files to unauthorized users' repositories. Unfortunately, this would impose restrictions on repository synchronization: Confidential data could not be distributed via unauthorized users or a central repository anymore. Further, the VCS would have to be able to deal with incomplete revision histories (if a user is not authorized for a revision on a path) and incomplete revisions (missing single files).

We therefore do not limit synchronization of confidential files at all but ensure confidentiality via encryption. To preserve compatibility with other repositories and code hosting platforms, we do not change any internal repository data structures. Instead, we extend management of working copies.

The basic idea is to separate the working copy into two layers: The *regular working copy* which is used during execution of any VCS operations and a *virtual view* on top of it. Only the virtual view contains confidential files as they are seen by users. In that view, a user can work with confidential files as if they were regular files. During commit, changed files are encrypted transparently and the resulting ciphertext and metadata for access control are added to the regular working copy. Accordingly, metadata/ciphertexts in the regular working copy are interpreted and decrypted during checkout, refreshing their counterpart in the virtual view. This way, confidentiality is achieved and both compatibility and simple usability are preserved: If support for confidential files is missing, encrypted contents appear in the working copy but no further restrictions apply; if support is given, the user sees no differences between regular files and confidential files she is authorized for. The relation between the two layers of the working copy is illustrated in Figure 2; the detailed mapping is described in the following subsections.

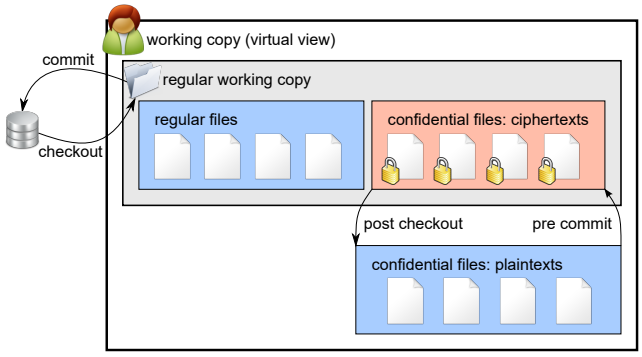

Fig. 2. Working copy is separated into two layers

1) Encryption of File Contents: Each confidential file in the virtual view is mapped to an encrypted file in the regular working copy whose name consists of a randomly chosen identifier and a prefix identifying its owner. Its content is encrypted with a symmetric cipher. In principle, any secure encryption scheme could be used, although regular, randomized schemes would cause storage overhead since they prevent the VCS from computing space-efficient differences between file versions.

To allow for storage efficiency, we adopt the scheme from [13]: Each file content is encrypted under a randomly chosen key tuple (K_R, K_O) which is made known to authorized users. K_R is the classic encryption key chosen to be unique for a specific combination of access rights (i.e., K_R is renewed when rights change) and K_O is a *convergence secret*, or *obfuscator* supposed to be unchanged in a file's lifetime. The scheme first splits a content deterministically into non-overlapping, dynamic-size chunks by using the content-defined chunking (CDC) approach of Muthitacharoen et al. [15]: A w-bytes sliding window is moved over the content and all positions with a hash value within a certain range are declared chunk boundaries. To prevent boundaries from leaking information about the content, the used hash function is keyed by K_O. The ciphertext of a file is defined as the concatenation of its individually encrypted chunks, each chunk being encrypted deterministically. This way identical plaintext chunks are mapped to identical ciphertexts, allowing the VCS to compute deltas. The scheme uses convergent encryption (CE) as introduced by Farsite [8] for this purpose: A chunk is encrypted using its hash as its key, and the key itself is encrypted with K_R to allow decryption. The convergence secret K_O is included in the hash computation to thwart known-plaintext attacks.

2) Encryption of File Names: Storage saving potential of data deduplication mechanisms w.r.t. file names is low. Therefore we encrypt file names with a common CCA-secure, symmetric encryption scheme. The resulting ciphertext is stored in a metadata file in the regular working copy. Note that we cannot prevent different users from creating confidential files with identical names without revealing information. Thus, we resolve potential conflicts locally in a user's virtual view.

3) Management of Read Access Rights: To allow read access to a confidential file, its key has to be distributed to authorized users. In [13], this is realized using an authenticated Diffie-Hellman key exchange whose protocol steps are realized using a sequence of commit/update operations executed by their involved users. In a distributed system, this approach would be impracticable, as additional push/pull operation executions—possibly involving several intermediate repositories—would be necessary, causing considerable communication overhead. On the other hand, existing distributed VCS already have support for OpenPGP [6] certificates: Git has native support for signing commits using GPG [7, Chapter 7.4] and Mercurial comes with a corresponding extension [5]. Given this support, it seems natural to utilize OpenPGP for key distribution, too. We therefore require each user to be in possession of an OpenPGP certificate in order to access confidential files. A file owner can then grant rights by encrypting the necessary keys using the authorized user's public OpenPGP key. Encrypted keys are stored in a metadata file of the confidential file in the regular working copy.

To ease key distribution, only K_R is explicitly distributed this way. The corresponding obfuscator K_O is encrypted with K_R and prepended to the ciphertext. This ensures that every write-authorized user is able to change K_O without requiring changes to metadata files (which would have to be signed by the respective file owner).

C. Authenticity Concept

In a distributed VCS, modifications of confidential files cannot be entirely prevented as an attacker is always able to modify her local data structures. Our goal is to prevent *undetected* modifications in working copies of authorized users. Verifying changes during push/pull would impose similar problems as discussed for confidentiality. Additionally this would add considerable requirements to the authenticity verification mechanism: If unauthorized users or a central server should be used as a proxy, they would need to be able to verify authenticity. From a security perspective, however, it is sufficient if authorized users are able to do so. For that reason and with compatibility in mind, we leave push/pull unchanged.

As described in Section II, authenticity should be ensured in several respects: File contents may be modified by authorized users, names and other metadata only by a confidential file's owner. We now discuss how we achieve these goals.

1) Authentication of File Contents: Whenever a user changes a confidential file's content, we require her to compute a signature on next commit to prove its authenticity. To prevent replay attacks, the signature is computed for the combination of content, associated metadata and base revision number.[1]

For being able to verify authenticity of file contents, authorized users must know which other users are in possession of write access rights, i.e., which public keys correspond to valid signatures. The corresponding information is maintained by the file owner and stored in the confidential file's metadata.

2) Authentication of File Names and Metadata: Metadata including a confidential file's name and access rights may only be modified by its owner. Their authenticity is ensured in a straightforward way: Whenever data is changed, the file owner signs the resulting data set. Again, we include the base revision in the signature computation to prevent replay attacks.

3) Authentication of File Deletion: If any user was permitted to delete a confidential file, an attacker would be able to replace it with a non-confidential one she has access to. Unless this is noticed by authorized users, she could gain access to confidential information that users commit afterwards. Similar to other metadata changes like file renaming, deletion of confidential files must, thus, only be possible for their owners.

Deletion of confidential files requires special treatment, though, as it involves deletion of metadata files that carry information used for authenticity verification. We circumvent this problem using *lazy deletion*: If an owner deletes a confidential file, the file is tagged as *deleted* in its metadata file, but not actually deleted from the regular working copy. Clients of authorized users can verify authenticity of the deletion analogously to other metadata changes and hide the file from the virtual view. Such semi-deleted files are completely removed from the regular working copy during the next commit.

4) Management of Access Rights: Every user authorized to access confidential files needs one OpenPGP certificate that is used for signing and key distribution. We describe integration of OpenPGP into our concept in detail now.

a) Identities and Trust: First, we need to correlate VCS users and their OpenPGP keys. Distributed VCS identify users based on their name and email, but without any verification. Public OpenPGP keys, on the other hand, can be identified via an ID, an associated user ID (typically "Firstname Lastname (Comment) <Email Address>"), or a key fingerprint. We use the email address to match VCS users and their OpenPGP certificates. We establish trust between users based on OpenPGP's web of trust as follows: We require bidirectional trust between a file's owner and any authorized user, but no immediate trust between pairs of users of a file as this would likely be impractical. Instead, trust is established transitively via the owner: All authorized users' certificate fingerprints are included in the access right lists signed by the file owner. Using these fingerprints, users can fetch the corresponding full certificates from centralized OpenPGP key servers and rely on their authenticity due to the owner's signature. Given these certificates, they can verify content signatures of any authorized user—requiring explicit trust only in the owner's certificate.

b) Granting and Revocation of Access Rights: For confidential files with different users, a variety of metadata are generated and stored. The representation of a confidential file at a specific point of time is illustrated in Figure 3. Rights have to be granted by the file owner to be valid and can change over time. The process of granting/revocation of rights is described below.

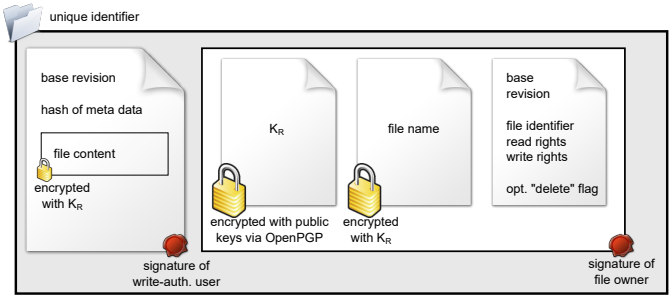

Fig. 3. Representation of confidential file in regular working copy

To grant a user a read right, the file owner essentially has to encrypt the file's key K_R with the public OpenPGP key of the user. Meeting the requirements from Section II requires extra efforts, though, as users should only be granted rights for specific file revisions. Every access right change thus requires renewal of K_R. In detail, the following steps are performed by the file owner's client when granting a read access right:

1) The user and her OpenPGP certificate fingerprint are added to the list of read rights.
2) A new random key K_R is created.
3) K_R is encrypted using OpenPGP for each individual read-authorized user using her OpenPGP key.
4) The file name is re-encrypted using K_R.
5) The current revision is set as base revision in metadata.

[1]Precisely, the concatenation of these data's hash values is signed. Note that we could not use the *resulting* revision's number instead of base revision as it depends on all contents of the revision, including the signatures.

6) Changed metadata is signed using u_o's OpenPGP key.
7) File content is re-encrypted[2], base revision and metadata hash are updated, and the combination is signed by u_o.

Revocation of read access rights is performed analogously.

When a write right is granted, the user and her certificate fingerprint are added to the list of write rights similar to Step 1. As a write right only makes sense in combination with read access, Steps 1 to 7 are executed to also grant *read* access if necessary. Otherwise, only Steps 5 to 7 are executed to prove metadata authenticity. Revocation is performed analogously.

5) Authenticity Verification: A confidential file is authentic if *all* operations in its revision history are authentic. Verifying authenticity of the whole history on each checkout would not scale, though, as any further revision caused additional overhead. Since revisions are immutable, it is sufficient to perform this verification once per revision and client and store the result at a place not synchronized between repositories (to ensure that a client trusts only its own decisions). We perform the following steps on checkout of a revision:

1) Check in local database (LD) whether base revision(s) have already been verified and verify them if necessary.
2) For each new, updated or deleted confidential file:
   a) If file has been removed, check whether the *deleted* flag was present in its last revision's metadata.
   b) If metadata have changed:
      i) Verify metadata signature. If correct, let F_o be the used OpenPGP certificate's fingerprint.
      ii) Determine file owner. Let Id_o be her identity.
      iii) Verify if revision was created by Id_o according to the VCS.
      iv) Verify whether OpenPGP certificate with fingerprint F_o for identity Id_o is trusted.
      v) Verify whether base revision and file identifier mentioned in metadata are correct.
   c) If file content has changed:
      i) Verify whether content signature is correct. If so, let F_{rw} be the used OpenPGP certificate's fingerprint.
      ii) Let Id_{rw} be the identity of the user that created the revision including the change.
      iii) Verify whether combination of F_{rw} and Id_{rw} is present in the file's write access rights.
      iv) Verify whether base revision and metadata hash mentioned in content metadata are correct.
3) In case of success, store verification result in LD.

Confidential files with correct metadata signatures created using untrusted OpenPGP certificates are hidden from a user's virtual view, so files of owners she does not trust are invisible to her. If other verification steps fail, it stands to reason that the file has been tampered with, so we abort checkout then.

[2]The encryption scheme borrowed from [13] ensures that delta computation on ciphertexts remains possible even if access rights (and thus K_R) change. As discussed in the source, this has a slightly negative effect on confidentiality, as unchanged fragments (≈ 256 bytes) across revisions remain recognizable due to deterministic encryption. Write-authorized users can change the *obfuscator* at any time to hide this information; changing it for each revision results in CCA-secure encryption. Details are covered in the source.

Three special cases have to be considered, though. First, special handling is required if the user is owner of a confidential file: If she does not trust her own certificate, malicious modification has to be assumed, too, so we also abort checkout in that case. Second, revisions created via merge have two base revisions. A checkout is successful if *both* base revisions are authentic and if each confidential file is the result of legitimate changes with respect to *any* base revision. Third, revisions have to be rechecked for authenticity if a user starts trusting an owner whose files were previously hidden from her virtual view.

D. Limitations

The authentication concept imposes a significant restriction on *merges*: If confidential files are modified in different branches that should be merged together, the merging user has to be authorized for the change resulting from the merge, i.e., she has to be file owner to merge metadata and needs write access to merge modified contents. If multiple files are involved, the restrictions might conflict, e.g., if Alice (Bob) is only authorized to change A (B) as in Figure 4, neither of them would be allowed to merge X/Y. This has to be resolved manually by splitting the merge as shown in the figure.

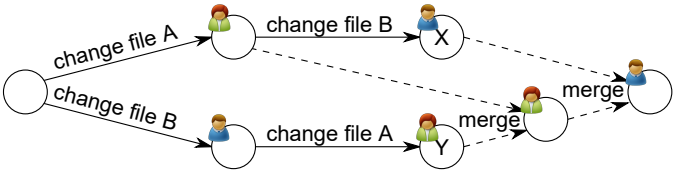

Fig. 4. Merge of revisions with conflicting changes in different confidential files

IV. The Mercurial Extension

As part of our work, we implemented the concept prototypically into *Mercurial*, which was selected due to its easy extensibility: Mercurial is written in Python and supports loading of extension modules that can hook into its control flow, extend existing operations and define entirely new operations. Existing data structures, e.g., the repository, can be accessed from extension modules in an object-oriented way.

Our extension uses several Python modules: *PyCrypto* [14] provides cryptographic operations, *PyYAML* [19] is used for config/metadata storage, and the wrapper *python-gnupg* [18] realizes integration of the OpenPGP implementation GnuPG. We had to modify the latter to allow consideration of user IDs in addition to OpenPGP keys for trust relationship verification.

The Mercurial extension is available for download under https://github.com/michaellass/hgcrypt and described below.

A. Storage Structure

Mercurial stores the user's working copy in a regular folder on the user's system. A hidden subfolder `.hg` contains the repository, configuration files etc. The newly introduced confidential files, i.e., their encrypted contents and metadata, should be under version control as if they were a regular part

of the user's working copy. However, these files should not be modified manually and ideally not even be seen by the user (the virtual view should hide the regular working copy).

To achieve this, we introduce a second hidden subfolder `.hgcrypt` with two subfolders in it, `private` and `public`: `public` contains the data that should be included in the repository, i.e., it is treated like a regular directory in the user's working copy. For each confidential file it has a subfolder, containing its encrypted content, metadata and corresponding signatures as shown in Figure 3. `private`, in contrast, is excluded from *all* common operations[3] (thus not included in the repository) and contains private, only locally available data—from now on referred to as *global metadata*. Here, configuration data and security-critical information about decrypted confidential files, results of signature verifications, and temporary data like pending changes in the user's virtual view are stored. Details about the contents are listed in Table I.

Data entry	Description
`fingerprint`	fingerprint of user's OpenPGP certificate
`id-name-mapping`	mapping between decrypted file names (virtual view) and confidential file identifiers (regular working copy)
`ignored-files`	list of all files that should be ignored by Mercurial, i.e., excluded from synchronization with repository (includes `.hgcrypt/private/*` and decrypted confidential files in virtual view)
`hashes`	hashes of decrypted files used for change detection
`to-be-added/ -moved/-deleted`	additions, renames, deletions pending for next commit
`to-be-merged`	files added due to merge, including base revision
`new-perms`	pending access right changes
`to-be-obfuscated`	pending obfuscator changes
`verified`	list of successfully verified revisions

TABLE I
INFORMATION STORED IN GLOBAL METADATA

B. Operations

Our extension changes the configuration of Mercurial such that global metadata and confidential plaintexts are ignored. All other changes concern individual operations.

1) Modified Operations:

- *add* puts files under version control. We prevent decrypted representations of confidential files from being added as regular files and introduce a parameter `-p` forcing creation of confidential files, involving generation of a random identifier, storage of relations between file names and identifiers in global metadata and inclusion in the `to-be-added` list.
- *rm* removes files. We enable file owners to initiate lazy deletion of confidential files as described in Section III-C3.
- *mv* allows to move or rename files. We extend it to account for virtual view representations of confidential files: Moves performed by file owners are included in the pending changes (`to-be-moved`) and executed during next commit.
- *checkout/update* checks out a specific revision. We ensure the original operation is only executed if the revision's authenticity has been successfully verified. Verification is performed according to Section III-C5 and results are stored for future checkouts.

At the end of checkout, changes to confidential files are evaluated: Decrypted files are removed from (created in) the virtual view if confidential files are deleted (added) or the user lost (got) read access rights. Decrypted files are updated if their ciphertexts have changed or if a confidential file has been moved/renamed. Global metadata are updated accordingly and file name conflicts are resolved locally by adding suffixes.

- *merge* merges changes from another development branch (revision Y) into the revision X currently checked out. It determines the most-recent mutual ancestor revision C of X/Y and applies the changes from C to Y to the working copy.

If confidential files are affected, we require Y to be authentic as described before. Afterwards, we execute the original merge procedure which covers the encrypted representations of confidential files. Changed confidential files are stored in `to-be-merged`, and—if possible—decrypted to create/update virtual view representations. On next commit, further changes to these files are detected based on the list and encrypted representations are updated before being stored in the repository. Note that our implementation currently requires manual conflict resolution for merges involving conflicting changes to the same confidential file (see Section III-D).

- *commit* synchronizes local changes to the repository. As the original commit operation already covers synchronization of confidential files provided that their encrypted representations are included in the user's regular working copy, we only have to ensure that changes inside the virtual view are transferred into the regular working copy beforehand. For this, we re-encrypt changed file contents (which we identify based on `hashes`), apply changes from the pending changes lists and update/reset all lists accordingly.

Pending changes are processed as follows: New confidential files (`to-be-added`) are initially encrypted, signed and stored under their identifiers in `.hgcrypt/public`. Encrypted representations of deleted files are removed using *lazy deletion* (see Section III-C3): First, residues of files already marked *deleted* are removed. Second, contents of freshly deleted files (`to-be-deleted`) are deleted, a *deleted* flag is stored in their metadata, and the results are signed. New names of moved files (`to-be-moved`) are encrypted, signed and stored in the files' metadata.

Changed rights are stored in the file's metadata, too. In that case a new random K_R is assigned, implying re-encryption of file name and content, key distribution to the new set of users, and re-signing of the file's content and metadata. Obfuscator changes imply re-encryption / re-signing of file contents, too.

We also integrated appropriate handling for confidential files into *status* and *revert*. No changes to *push*/*pull* were required.

2) Added Operations (specific to confidential files):

- *setacl* allows file owners to administer access rights of their confidential files. The operation determines the PGP key of the *target user* whose rights are to be changed, possibly requiring to select one of several available keys or to enter a fingerprint for retrieval from a key server. Granting of rights

[3]Exclusion from all operations is an essential security aspect: If, e.g., a checkout was allowed to write to this directory, an attacker might obtain access to confidential files by checking in tampered files in this directory that overwrite configuration data of a benign user's Mercurial client.

is only possible if the combination of target user identity and determined PGP key is trusted; revocation is allowed in any case as long as the target user differs from the file owner.
listacl lists access rights for a file, *lscf* provides details about confidential files present in a user's virtual view and *obfuscate* schedules a change of a file's *obfuscator* for the next commit.

C. Limitations

Note that Mercurial provides further operations (e.g., *log*, *diff*) which remain usable, but require additional implementation efforts from a usability perspective, i.e., to work transparently on decrypted representations of confidential files.

Another limitation applies to our use of GnuPG: Due to lack of native support for verifying trust relationships with respect to a combination of key and email address, we use an operation that lists all user IDs and corresponding trust statuses for a specific key. Unfortunately, this operation only shows the *current* trust status and does not allow to determine the trust status with respect to a specific point of time. Since existing revisions and associated metadata are immutable, confidential files will be rendered unusable as soon as their owners' keys expire or are explicitly revoked. A slight modification of GnuPG would clearly fix that issue.

V. Security Evaluation

In this section, security is analyzed with respect to the goals stated in Section II.

A. Confidentiality of File Names

Consider a *passive* attacker u_{na} who wants to break confidentiality of file names. A repository contains a confidential file's name only in an encrypted representation as part of its metadata, created using a CCA-secure encryption scheme with a symmetric key K_R that is in turn stored encrypted with the public GnuPG keys of all read-authorized users. K_R is changed with every read access right change, so u_{na} cannot gain further information about K_R from other revisions. To decrypt the file name, u_{na} would either have to break one of the encryption schemes (which are assumed to be secure), get a user's private GnuPG key (which would elevate her to u_r according to our security model), or guess K_R via brute force. The latter is practically impossible as K_R is sufficiently long (256 bits in our implementation) and chosen at random by u_o whose system is assumed to be benign.

An *active* attacker could further try making a user u_o tell her the file name, e.g., by encrypting it with a key K_R that has previously been tampered with by the attacker. This requires changing the files' metadata (where K_R is stored), though, and therefore breaking authenticity of the files' metadata.

B. Confidentiality of File Contents

Regarding confidentiality of file contents, the same arguments as for file names apply. Since contents are encrypted with the scheme from [13] instead of a CCA-secure one, though, only CDPA$_d$-security ($d = \min\{w, l\} - 1$, where l is the minimum chunk size that we set to the rolling hash window size w) is achieved. As long as the obfuscator remains unchanged across access right changes, the security guarantees further apply only to attackers u_{na}. Users $u_{r'}$, as discussed in [13], are in possession of the obfuscator and thus able to verify existence of already known chunks in the encrypted file.

C. Authenticity of File Contents

Assume a file content was tampered with by a user u_r and the modified file content is present in another user's working copy. Section III-C5 implies that the modified file content must have been signed with a GnuPG key with write access according to the file's metadata. Assuming authenticity of the metadata, the attacker u_r must either be in possession of the private GnuPG key of a write-authorized user (elevating her to a user u_{rw} according to our security model) to create the signature, or she must have replayed an existing one. The latter is possible, but only if the corresponding file content is copied, too, and if the base revision x_b of the new revision x_t matches that of the revision x_s the signature was copied from. Then, however, the attacker has only created a copy of an existing branch of the file. As the modification leading to x_t was technically created by a user u_{rw} (who created x_s from x_b) and not by u_r (who created x_t from x_b), this is not an attack according to Section II.

D. Authenticity of File Metadata

Assume metadata of a confidential file has been tampered with by u_{rw} and this modification is present in the working copy of a user u without being noticed. Provided that the attacker u_{rw} does not know the file owner's private GnuPG key (which would elevate her to u_o) and assuming that GnuPG's signature scheme is secure, the only option for u_{rw} to create valid signatures is to create an own GnuPG key pair with the actual owner's user ID.

If u is the file owner, this situation could not have occurred, though, since the client assumes a malicious modification (see Section III-C5) if a signature is detected that has been created using a certificate containing her user ID but a foreign key. A non-owner u cannot detect such a malicious signature, but the file would be hidden from her working copy unless the combination of used GnuPG key and user ID is trusted according to her web of trust view. Given that trust, the change would be a valid change of u_o from her point of view.

The only remaining option for u_{rw} to generate a valid signature would be to copy it from another revision. As signatures are computed over the whole metadata including base revision, u_{rw} can only clone modifications actually done by u_o, which is not an attack according to our threat model.

VI. Performance Evaluation

A. Storage Efficiency

As we adopted the encryption scheme from [13] for storage efficiency, we rely on our previous evaluation results and apply convergent encryption (CE) to chunks created via content-defined chunking (CDC) with 256 bytes avg. chunk size and rolling hash window and min. chunk size set to 48 bytes. To

verify that the results remain valid in the Mercurial setting, we repeated the real-repository evaluation of [13], i.e., we re-enacted a part of the revision history of ispCP[4] [1] to measure storage overhead. We started with empty repositories and committed changes of the original repository, ignoring metadata like commit messages as they are out of scope of our extension. Results are shown in Figure 5: Unsurprisingly, a regular repository (solid blue line) has lowest storage consumption as Mercurial can apply data deduplication and compression. Encrypting changed file revisions using a regular scheme (advanced encryption standard (AES) in cipher block chaining (CBC) mode with random initialization vectors and static key) prevents either of them, leading to 6 times higher costs (purple). Using our extension for user and key management on top of that scheme incurs only little additional overhead (red). The solid green line finally shows the savings achieved by CDC and CE: Only half of the storage overhead w.r.t. the unencrypted repository than with regular encryption is caused, and costs are considerably lower than those for using encryption without our extension.

For reference, we also plotted the corresponding evaluation results for SVN: The dotted lines show that relations between the different experiments are comparable to those in the SVN evaluation, and that Mercurial is more storage-efficient than SVN in any case thanks to its different storage concept.

B. Computational Overhead

We evaluated our implementation's performance based on files taken from the Linux kernel repository [2] (Table II). Starting with tag `v4.4` we chose two sets of files, each consisting of files of similar size in the most-recent revision, and extracted the last 100 changes made to each file. The table shows average values over these revisions. Measurements were performed on a single core of an Intel i5-3210M with data located in memory. We used a Rabin-fingerprint-based CE implementation (see [13]).

Set	Path	Size (KiB)	Lines of code
1	arch/ia64/kernel/acpi.c	24.8	1017
	drivers/pcmcia/pcmcia_resource.c	25.5	994
	kernel/time/tick-broadcast.c	19.0	751
	lib/swiotlb.c	25.5	925
	net/core/net_namespace.c	14.4	636
	net/mac80211/agg-tx.c	22.9	796
	sound/pci/hda/patch_conexant.c	99.7	3390
2	drivers/gpu/drm/i915/intel_display.c	432	15698
	drivers/net/ethernet/broadcom/bnx2x/bnx2x_link.c	392	13474
	drivers/net/ethernet/broadcom/bnx2x/bnx2x_main.c	395	14589
	drivers/net/ethernet/broadcom/tg3.c	461	18102
	drivers/net/wireless/ipw2x00/ipw2200.c	326	12121
	drivers/scsi/lpfc/lpfc_sli.c	502	16434
	net/wireless/nl80211.c	333	12666

TABLE II
FILE SETS USED FOR PERFORMANCE EVALUATION

[4]The data set is rather varied: On average, each revision affects ≈ 25 files, each of which has ≈ 2.8 clusters containing changed content. StDev of files affected per revision is ≈ 131 and StDev of changed clusters per file is ≈ 4.7.

1) Commit: For each file set, we committed 100 changes to a Mercurial repository and measured the total execution time. Measurements were repeated 50 times each for three different scenarios: A conventional repository, a repository with our extension enabled but using non-confidential files, and a repository in which all files are marked confidential.

Results are shown in Table III. By comparing them with times required for committing only a single file from a set, we dissected the time for committing a single revision into a static part and a part per confidential file: Loading the extension adds about 50ms to each commit. Additional time for each confidential file depends on its size, but with 150ms for relatively large source files it can be rated barely noticeable.

2) Checkout: For an initial checkout we evaluated the time required to verify all 100 previously committed revisions (see Section III-C5) containing confidential files. As this verification only takes place if confidential files have been changed, only the scenario where each file is marked as confidential was considered. Again, measurements were repeated 50 times.

Results are shown in Table IV. Again we dissected the measured time into a static and a per-file portion. Verification of a single revision takes about 1ms with additional 23ms for each confidential file; file sizes have negligible impact. Verification time can therefore be assessed as only noticeable during an initial checkout involving many revisions.

Set		100 revisions Time (s)	100 revisions StDev (s)	Single revision Static (ms)	Single revision Per File (ms)
1	Unmodified Mercurial	8.30	0.074	61.4	3.1
	Extension loaded	13.56	0.099	113.0	3.2
	Commited as confidential	51.92	0.422	114.5	57.8
2	Unmodified Mercurial	14.13	0.125	63.1	11.2
	Extension loaded	19.29	0.134	115.8	11.0
	Commited as confidential	116.28	0.954	116.7	149.4

TABLE III
TIME REQUIRED FOR COMMIT

	100 revisions Time (s)	100 revisions StDev (s)	Single revision Static (ms)	Single revision Per File (ms)
Set 1	16.16	0.168	0.7	23.0
Set 2	16.51	0.162	1.2	23.4

TABLE IV
TIME REQUIRED FOR VERIFICATION

VII. RELATED WORK

Except for the SVN extension [13] on which our work is based, we are not aware of any further research in the special field of VCS security. Challenges and solutions similar to ours can be found in the related field of secure file systems, though:

SiRiUS [11] and Plutus [12] are examples of file systems that support file-level access rights that are enforced via encryption. Similar to our solution, SiRiUS encrypts each file symmetrically using a randomly chosen key and distributes it to authorised users by encrypting it using their public keys. Plutus further deals with efficient key distribution in presence

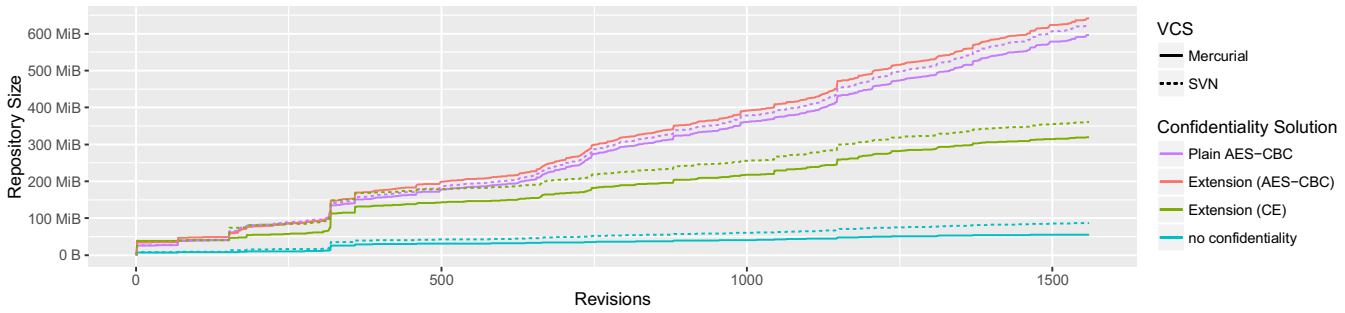

Fig. 5. Storage costs of Mercurial extension in comparison to [13]

of many users and frequent right changes: Files with identical rights are grouped into *filegroups* and their keys are collected in a corresponding *lockbox* so that distribution of a single filegroup key is sufficient to grant access to multiple files. *Key rotation* (later replaced by *key regression* [10]) ensures that a file key allows access to all prior versions. *Lazy revocation* eliminates the need for immediate re-encryption of files after right changes: When keys have to be changed, re-encryption is postponed until the next write access—similar to our *lazy deletion* concept. Since semantics of rights are different in our solution (rights are bound to revisions and should not imply access to previous revisions), we employ the SiRiUS approach.

Convergent encryption, the essential component for our solution's storage efficiency guarantees, was initially introduced by Douceur et al. [8] as part of *Farsite*. Its goal is to allow deduplication of identical files of different users despite encryption. Storer et al. [20] were the first to apply CE to chunks produced by *CDC* as to allow deduplication for similar files. Tahoe-LAFS [21] uses CE at the file level like Farsite, but introduces a *convergence secret* as to improve its security guarantees at the cost of slightly worse deduplication efficiency. We adopted the combination of both extensions proposed and proven secure in [13], i.e., we use a convergence secret to improve security, but perform CDC beforehand.

All works have in common that they propose an isolated, new system, while we aim at enabling an easy transition from an insecure, established system to one with strong security.

VIII. Conclusion

We have presented a concept for secure and efficient storage of confidential files in distributed VCS. It allows legitimate users of a repository to manage read/write rights for individual files, which are effectively enforced using field-tested cryptographic measures: Read access control is achieved by symmetric encryption of file contents and names; integration of signatures allows users to verify authenticity of file contents and metadata with respect to write access rights. In contrast to the use of standalone encryption tools, however, access control is seamlessly integrated into the VCS and its use is transparent to the user: Once the system is set up, users take advantage of strong security properties without having to think about (or even noticing) that cryptography is in place, except for a slight performance degradation. Storage overhead is minimized by a specialized encryption scheme based on chunking and convergent encryption, which allows delta computation on ciphertexts as required by the VCS. The concept has been proven secure and has been implemented for Mercurial in a way that it is compatible with unmodified Mercurial versions and code hosting platforms.

References

[1] http://www.isp-control.net:800/ispcp_svn/trunk.
[2] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/.
[3] Apache Software Foundation, "Apache Subversion," http://subversion.apache.org/, 2015, accessed 2016-01-06.
[4] Atlassian, Inc., "Bitbucket — The Git solution for professional teams," https://bitbucket.org/, 2016, accessed 2016-01-25.
[5] B. Benissot, "GPG extension," https://www.mercurial-scm.org/wiki/GpgExtension, 2016, accessed 2016-01-06.
[6] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, "OpenPGP Message Format," RFC 4880 (Proposed Standard), Internet Engineering Task Force, Nov. 2007, updated by RFC 5581.
[7] S. Chacon and B. Straub, *Pro Git*, online ed. New York: Apress, 2009, http://git-scm.com/book, accessed 2016-01-06.
[8] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer, "Reclaiming space from duplicate files in a serverless distributed file system," Microsoft Research, Technical Report MSR-TR-2002-30, 2002.
[9] Free Software Foundation, "CVS – Concurrent Versions System," http://www.nongnu.org/cvs/, 2006, accessed 2016-01-06.
[10] K. Fu, S. Kamara, and T. Kohno, "Key regression: Enabling efficient key distribution for secure distributed storage," in *Proc. of NDSS*, 2006.
[11] E. Goh, H. Shacham, N. Modadugu, and D. Boneh, "Sirius: Securing remote untrusted storage," in *Proceedings of NDSS*, 2003.
[12] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *Proceedings of FAST*, 2003, pp. 29–42.
[13] D. Leibenger and C. Sorge, "A Storage-Efficient Cryptography-Based Access Control Solution for Subversion," in *Proc. of SACMAT*, 2013, pp. 201–212.
[14] D. Litzenberger, "PyCrypto - The Python Cryptography Toolkit," https://www.dlitz.net/software/pycrypto/, 2015, accessed 2016-01-06.
[15] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *Proc. of SOSP*. ACM, 2001, pp. 174–187.
[16] B. O'Sullivan, *Mercurial: the definitive guide*, online ed. Sebastopol, CA: O'Reilly Media, Inc., 2009, http://hgbook.red-bean.com/read/.
[17] M. Rochkind, "The source code control system," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 4, pp. 364–370, Dec 1975.
[18] V. Sajip, "python-gnupg," https://bitbucket.org/vinay.sajip/python-gnupg/, 2015, accessed 2016-01-06.
[19] K. Simonov, "PyYAML," http://pyyaml.org/wiki/PyYAML, 2014, accessed 2016-01-06.
[20] M. W. Storer, K. Greenan, D. D. E. Long, and E. L. Miller, "Secure Data Deduplication," in *Proc. of StorageSS '08*. ACM, 2008, pp. 1–10.
[21] Z. Wilcox-O'Hearn and B. Warner, "Tahoe: the least-authority filesystem," in *Proceedings of StorageSS '08*. ACM, 2008, pp. 21–26.