Secrecy in concurrent version control systems

Jerônimo Pellegrini

[1]Instituto de Computação – Unicamp

jeronimo@ic.unicamp.br

Abstract. *This paper describes two cryptographic protocols for incorporating secrecy in concurrent version control systems in such a way that neither text nor passphrases are ever sent to possibly hostile servers. One of the protocols works for centralized and one for distributed version control systems. Most operations, as defined by the protocols, take linear time on the size of keys or the size of changes made to the content, and the most frequent ones do not depend on the number of users. Both protocols rely on a public key infrastructure for access control.*

1. Introduction

The development of programs and documents by groups of people is often done using version control systems (VCSs). Today these tools play a central role in software configuration management [7, 3]. Current VCSs have authentication mechanisms and support secure transmission of data between client and server (or between peers in distributed systems), but to the best of our knowledge none of them offer a good degree of secrecy so that they can be installed on a hostile server. A recent survey on the development of software configuration management by Estublier and others [3] does not mention secrecy or encryption; no article in the proceedings of the 12^{th} International Workshop in Software Configuration Management mentions secrecy either [6]. However, there are situations where secrecy is desirable:

- The version control system is hosted by a specialized third party. The company may trust this third party's technical competence, but not want to share important secrets with them;
- The company has competent system administrators, but does not want them to know more than the necessary in order to perform their job;
- If the hosts with the versioned content are stolen, the secrets in them could be used by a malicious third party.

This paper describes two protocols that bring secrecy to version control systems. The structure of this paper is as follows: Section 2 gives an overview of version control systems; Section 3 describes two protocols for incorporating secrecy into VCSs; Section 4 describes the implementation of a prototype for distributed VCSs; finally, Section 5 presents our conclusions and future work.

2. Concurrent version control systems

The basic goal of concurrent version control systems is to allow different users to work on the same content, while keeping a history of changes and allowing users to undo changes

if necessary. If two users make changes to the same part of the content, the system can at least notify them of the conflict. Version control systems usually have a client/server architecture. Before we explain the possible variations on this basic paradigm, there are some important concepts related to version control systems that need to be presented:

- **Repository**: this is where all content, along with the versioning meta-data, is stored. The repository may be centralized, replicated or distributed;
- **Working copy**: the directory tree with a copy of the content in order to make changes before sending them back to the repository;
- **Revision**: also called *version*. This is usually a natural number or a string composed of numbers and separators (like "2.1.13") that identifies the version of a file. It may also be a cryptographic hash (as in Monotone[9], for example). There is usually a partial order among these numbers (the order is partial when we have "branching" of the content);
- **Delta**: the differences between two revisions of the content. This may be a traditional Unix *diff* (a "patch"), or an xdelta [8], which work for binary data. The delta between two revisions is usually composed of the deltas between the revisions of the files. Some systems are able to compute the deltas of binary files and filesystem metadata, while others are not;
- **Branch**: sometimes it is necessary to split the content into different "branches". The same content tree is (logically) duplicated in the repository, and changes are made to each of them separately. For example, several software developers are working on a program. One of them may need to implement new features that impact the core of the program. This may be done in a new branch (so it will not impact the work of other developers), while the others work on the "main" branch. Later it is possible to incorporate changes from one branch into another;
- **Head**: when several users are working on different branches, there are several different revision sequences from the first version to the versions being worked on by different users. The latest versions in each branch are called "heads" (in some systems there is the notion of "heads of a branch", but for the sake of simplicity we don't use it);
- **Merge**: the different copies of the content will eventually need to be merged and turned into one. There are a number of different algorithms for merging all versions. These algorithms can identify and to some extent resolve conflicts between the different versions (for example, if two users make changes to the same file, but in different points, then both changes can be applied, one at a time).

Version control systems usually implement different paradigms (all of them derived from the basic idea presented before). However we were not able to find a categorized description of all of them. We have decided to briefly describe some of the characteristics of version control systems that are relevant to this work.

- **Centralization**: the system may be *centralized*, *replicated* or *distributed*, depending on the possibility to have different data repositories storing different versions of the same content. Distributed version control systems are more recent, and new merging algorithms have been developed for them [1, 9, 2, 4]. Even if repositories are distributed, it is convenient to have a set of hosts working as servers;

- **Concurrency control**: if two users try to modify the same piece of information, there is a conflict. The system may avoid conflicts by using a lock – this is called the *lock-modify-unlock* paradigm – or it may just identify these conflicts, and leave the resolution to the users (as in most open source and free systems) – this is the *copy-modify-merge* paradigm;
- **Data representation**: the system may represent, store and transmit data in different ways. One possibility is to store the original version of each file, and then store deltas that can be applied in sequence in order to obtain later versions of the files Another approach is to periodically store a "whole version" of each file, to avoid applying too many deltas. Also, some systems may store directory trees, links, and possibly other filesystem meta-data, while others are not able to do that;
- **Signing of changes**: some version control systems allow for the cryptographic signing of deltas, making it possible to trace changes done to the content. A public key infrastructure is required, and different users of the version control system may verify, in the history logs, who incorporated a specific change. This is particularly important in the open source and free software context, where the content is developed with the help of contributors that are not bound by a contract (as would happen in a software development business, for example). This is necessary because of copyright and patent concerns (it should be possible to quickly identify an offending piece of code), and also for security reasons (a delta that introduces a trojan horse would need to be signed by someone).

In the rest of this paper, we will deal with systems that represent data as an initial revision and a sequence of deltas. We will present one protocol centralized and one for distributed systems. The protocols work for both conflict handling methods, and integrate smoothly with systems that have support for cryptographic signatures, as the protocols themselves need a PKI.

2.1. Dynamics of version control systems

This subsection describes the dynamics of version control systems and defines some operations that are relevant to understanding the protocols. This is neither standard nor is it the only way to describe these systems, but it is enough for our purpose.

2.1.1. Centralized systems

We start with the framework implemented by open source systems like CVS [12] and Subversion [11]: these are centralized, use the copy-modify-merge paradigm, and store sequences of deltas.

- **Grant** and **Revoke**: a new user may be granted access to one project in the repository, and an existing user may have his access revoked in the same way;
- **Import**: one user sends an initial version of all the data to the server. All needed meta-data is also created and stored on the server;
- **Checkout** and **Update**: one user asks for a specific revision of the repository. It could be "the latest version", or some other, like "the version of yesterday at midnight", or "version 143", and so on. If the user did modifications to his own working copy, then the client will identify that and try to merge the changes done

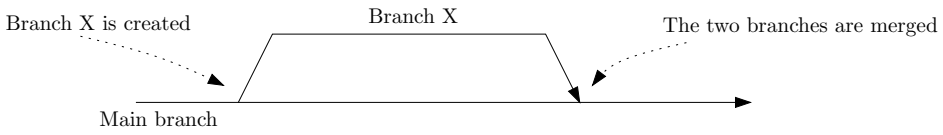

Figure 1. The creation of a new branch, and the later merge of the two branches.

by the user and the new version of the file that came from the server. If that is not possible, the client warns the user that a conflict happened. A destination path should be specified;

- **Checkin**: one user finished his modifications on the data, and sends to the server a delta that represents the differences between the latest revision (that is on the server) and his new version of the content. In most systems, this cannot be done if the user's working copy is outdated. However, the operation described here allows the user to specify a version on top of which the modifications will be applied. If the version was not the latest one, then a branch is created;
- **Rollback**: one user tells the server that a previous revision should now be considered the latest;
- **Branch** and **Merge**: given a branch b_1 and version v_1, a new branch b_2 may be created from there. Also, given two versions v_1 and v_2 of two branches b_1 and b_2, it is possible to merge both into b_1; Figure 1 illustrates this;
- **Delta**: given a repository, project and branch, it is possible to compute the difference between any two versions of the content. The same can be done using one version of the content in the repository and one working copy;
- **Apply**: a delta is applied to a working copy, and only later sent to the server.

There are other operations, like editing log entries, checking the history, and others. We have ignored most of them, since they can be treated the same way as checkins and checkouts.

Content repositories are often stored in trusted hosts, and the communication between the server and the users' workstations goes through an encrypted channel.

2.2. Distributed systems

This section briefly present distributed version control systems. Some examples are Monotone [9], Git [4] and Darcs [2].

These systems work just like centralized systems, except that instead of one central repository, there are several of them. A project may exist in a number of different hosts (usually one per user), and the deltas in each host may be different, since each user applies deltas to his local repository. Of course, there is a way to exchange the differences between two repositories, so that the project will have the same deltas on both of them. This is called *synchronizing*. The operations used in decentralized systems are the same for centralized ones, plus the following:

- **Synchronize**: given two repositories, a synchronization operation will exchange deltas between them so they will be identical at the end. This may result in a repository with multiple heads, as if branches had been performed (one can see the existence of different repositories as different branches of the project);

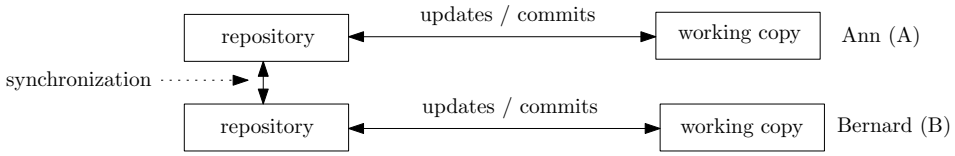

Figure 2. Two users with two repositories.

- **Merge**: merging algorithms for distributed version control systems are different from those for their centralized counterparts, since it may be necessary to merge multiple heads.

Figure 2 shows two users, each with his own repository (some systems hide the repository in the working copy, so the difference between them is not percieved by the user). Suppose Ann (A) and Bernard (B) have just synchronized their repositories so they have the exact same versions of the content. Then A and B start working independently. This will create divergence between their repositories: if there was a single sequence of revisions from version to version before, then there will be two sequences that diverge from the synchronization moment. Now, A and B may synchronize repositories again later. This will cause the repositories to be identical again – but both repositories will have both sequences, with two different "heads". These two heads may be merged, and then there would be one single head again. After the repositories are synchronized, it will seem to both users as if a branch had been made (as in centralized version control systems). The difference is that the branches were not explicitly created, and were in different repositories (each user is not necessarily aware of what happens in the other user's repository). Since this may happen with several users, a directed acyclic graph should be used to represent the ancestry of revisions.

In this work, the term *shared repository* is used for the hypothetical repository that would be the result of synchronizing all users' repositories. Informally, it's the "complete" version of the content, with all changes in all different repositories.

Section 3 explains how these operations can be changed so that no host ever stores the data without being encrypted.

3. The protocols

The goal of the protocols is to allow the VCSs to work without ever letting data go in clear to shared repositories (which can be hosted on hostile servers). To achieve this, every user needs to encrypt everything that he sends, and decrypt everything that he receives from any servers. The content deltas are computed as usual, but they are encrypted afterwards. This is done using a symmetric cipher and one single key k, shared by all users, but not by the servers: the key is only used by someone that needs to have a working copy. If the key is compromised, a new key is selected, and all users are informed. A public key infrastructure is needed (it could be the same used for the signing of deltas, when available), and the symmetric key k is encrypted with the users' public keys.

The following subsections describe the notation used in this work, the common part of both protocols, and the version control operations for both centralized and distributed versions of the protocol. In the common part of the protocol, we use "shared

repository" for either the server (in a centralized system) or the collection of repositories, as mentioned before (for distributed systems).

Notation

In the specification that follows, ID_X is "a unique identifier for user X"; all single capital letters (e.g., A, B) except K denote users (clients) of the system, each of them having a working copy; $\mathcal{G}$ is the group of users; K_A is "A's public key"; $\mathcal{K}$ is a list of symmetric keys; $\mathcal{M}$ is a mapping from revisions onto natural numbers; $\mathcal{R}$ is the set of all revisions; δ_i is the i-th delta stored on the repository (and δ_0 for the initial version of the content); Δ^r is the set of all deltas in a repository r; $\mathcal{G}$ is a group of users; Σ is a shared repository; $|$ is used for concatenation; $\{m\}_k$ means "message m encrypted with key k"; $A \rightarrow B$ is "A sends a message to B".

The protocol assumes the existence of an administrative group of persons ($\overset{*}{\mathcal{G}} \subseteq \mathcal{G}$), with administrative privileges; a star above a capital letter denotes one user in this group ($\overset{*}{C}$, for example).

3.1. Access control

This subsection describes access control, which is identical in both protocols.

Authentication is left to the underlying version control system. In centralized systems, users authenticate against the server. In distributed systems, users rely upon a PGP-like PKI in order to have some degree of confidence on data origin (all distributed version control systems support cryptographic signing of deltas). However, the authentication of entities who sign the deltas depend on how users build their web of trust.

3.1.1. The key list

Each user will have a copy of the following:

- A list of users $\mathcal{G}$;
- A list of keys $\mathcal{K} = (k_1, k_2, \ldots, k_n)$, with all the symmetric keys used to encrypt the content;
- A mapping from keys onto revisions $\mathcal{M} : \mathcal{K} \mapsto \mathcal{R}$.

This information is only transmitted to the user after being encrypted with the user's public key.

Since at any moment it may be necessary to switch to a new symmetric key, all users should have a list of all keys used, and the revision number when they started being used. With this, it is possible to decrypt all the deltas, one by one, using the appropriate keys for each of them.

The key list can be stored in a special location along with the content being versioned. Exchanging protocol metadata can be done, then, by adding or changing files as if they were versioned also.

3.1.2. Access grant

A new user A may be granted access to a project. The user sends a request to the admin group for that project, and one of the admin group members will send the key list encrypted with the user's public key to the repository (where the user can read it).

$\mathcal{G} = \mathcal{G} \cup \{A\}$
$\overset{*}{B} \rightarrow \Sigma : \ \mathcal{M}, \{\mathcal{K}\}_{K_A}$

3.1.3. Access revocation

A user A may have his access to a project revoked. It does not matter if he can read the current version of the data (because he did that in the past anyway), but he should have no access to future versions. The admin group will, then, pick a new key k_{n+1}, and future patches will all be encrypted with it. This new key is encrypted with the remaining users' public keys, and stored in the repository as the latest key.

$\mathcal{G} = \mathcal{G} \setminus \{A\}$
$\mathcal{K}' = \mathcal{K} \cup \{k_{n+1}\}$
$\mathcal{M}' = \mathcal{M} \cup \{n+1 \mapsto k_{n+1}\}$
$\forall C \in \mathcal{G}, \ \overset{*}{B} \rightarrow \Sigma : \{\mathcal{K}'\}_{K_C} | \mathcal{M}'$

3.1.4. Asymmetric key expiration, revocation and compromise

If a user's asymmetric key expires (or if he decides to revoke it), and there is no reason to believe that the symmetric keys were compromised, the user presents a revocation certificate to an administrator and asks him to re-encrypt the key list $\mathcal{K}$ with his new public key. The administrator sends the new key list to the shared repository.

$A \rightarrow \Sigma : K'_A$
$\overset{*}{B} \rightarrow \Sigma : \{\mathcal{K}\}_{K'_A} | \mathcal{M}'$

However, if the user believes that his key has been compromised, we assume that all the symmetric keys that he had were also compromised. The user then should perform the usual procedure to revoke his old public key, but the procedure described in subsection 3.1.5 for symmetric key compromise also needs to be performed.

3.1.5. Symmetric key compromise

If a key $k_i \in \mathcal{K}$ is compromised, the best that can be done is to suspend all access to the project, pick a new key k'_0, and use it to re-encrypt all deltas, one by one, encrypt the new key with the public keys of all users and send the new keys and deltas to the shared repository.

$\mathcal{K}' = \{k_0\}$
$\mathcal{M}' = \{\forall i, \ i \mapsto k_0\}$
$\forall \delta_i, \ \overset{*}{B} \rightarrow \Sigma : \{\delta_i\}_{k'_0}$

$\forall A \in \mathcal{G},\ \overset{*}{B} \rightarrow \Sigma : \{\mathcal{K}'\}_{K_A}$
$\overset{*}{B} \rightarrow \Sigma : \mathcal{M}$

3.2. The protocol for centralized systems

This section describes the protocol we propose for centralized version control systems, showing how the operations in section 2.1 should be changed.

3.2.1. Import

When a user does an initial import, a first symmetric key k_0 is randomly generated, and all the content to be imported is encrypted with it. The key k_0 is then encrypted with the keys of all the users in the project group ($\mathcal{G}$). This first key list is then added to the versioned content, in a special location, before the import is actually sent to the server.

$A \rightarrow \Sigma :\ \{\delta_0\}_{k_0}$
$\forall B \in \mathcal{G}, A \rightarrow \Sigma :\ \{k_0\}_{K_B}$

3.2.2. Checkout/update

The server sends all the deltas that the user needs to reach the latest version of the repository. Note that this is different from what sometimes happens with version control systems, where the whole content is sent at once. Usually, the server can calculate the delta from one revision to another, but in this case, the content is encrypted, so the deltas need to be decrypted at the user's side.

Besides the deltas, the user will also receive the latest version of the key list and key mapping. In the following, A is an ordinary user, and B belongs to the admin group. This is how the user asks for $r2$, and his working copy is r_1:

$A \rightarrow \Sigma :\ \text{checkout}, ID_A$
$\Sigma \rightarrow A : \mathcal{M}, \{\mathcal{K}\}_{K_A}$
$\forall\ \delta_i,\ r_1 < i < r_2 \Sigma \rightarrow A :\ \{\delta_i\}_{k_{\mathcal{M}(i)}}$

If one is concerned about the number of deltas to be applied, it is possible to regularly update a new complete version of the content (for example, everytime after a number of deltas, or after a fixed number of bytes). Then the number of deltas to be applied would be limited to the distance to the next complete version.

3.2.3. Checkin

The user tells the server that he wants to commit. The server will only allow that if the user has the latest version of the repository (i.e., he may need to issue an update first). The user is only allowed to commit if he has the latest key list (so he may need to update it first). He will then encrypt his delta with the latest key k_n and send it to the server.

$A \rightarrow \Sigma :\ \{\delta_i\}_{k_n}$

3.2.4. Rollback

It is usually enough to just send a "rollback" instruction to the server, telling which old revision should it rollback to. The server will then ignore the deltas that were applied after that revision.

$A \rightarrow \Sigma :$ rollback$, r_i$

3.2.5. Branching and merging

Let us suppose that the current version of the content is i. A user A opens a new branch, called "branch X", and commits some changes. Another user, B, commits changes to the main branch. We have, then, two deltas that may be applied to version i:

- δ_{i+1}: applying this delta will result in the version of the main branch after the modifications made by B;
- δ_{X_0} applying this delta will result in the version of the new (X) branch, modified by A.

All that our protocol needs is that the revisions be identified uniquely so that we can maintain the mapping $\mathcal{M}$, so there is no need for any special provision for branching.

Merging the two branches mentioned above is also done by applying deltas to both branches, so it doesn't require anything special either.

3.2.6. Conflicts

We will discuss conflicts for two different version control paradigms:

Lock-modify-unlock In the *lock-modify-unlock* paradigm, conflicts never actually happen, but users may need to lock the content at the server. If a whole revision of the project is locked at each time, then the server only needs to remember who was the user who locked that revision. However, for more fine-grained locks a problem arises: since the lock may contain sensitive information (like who is working on what part of the project, or even the layout of the project's directory tree), the locks should not be stored in clear text on the server. We can only see one solution to this problem: the locks can be implemented as part of the content, so they will not be stored in clear text on the server and the client software becomes responsible for not modifying content that is marked as locked.

Copy-modify-merge In the *copy-modify-merge* paradigm, conflicts are identified and resolved, as much as possible, by the client, and not on the server. This is done at the user's workstation after the content has been decrypted – so conflict resolution in this paradigm is out of the scope of the protocol.

3.3. The protocol for distributed systems

In a distributed version control system, there will be two different instances of each repository: one clear and one encrypted. The encrypted repository will keep the access control

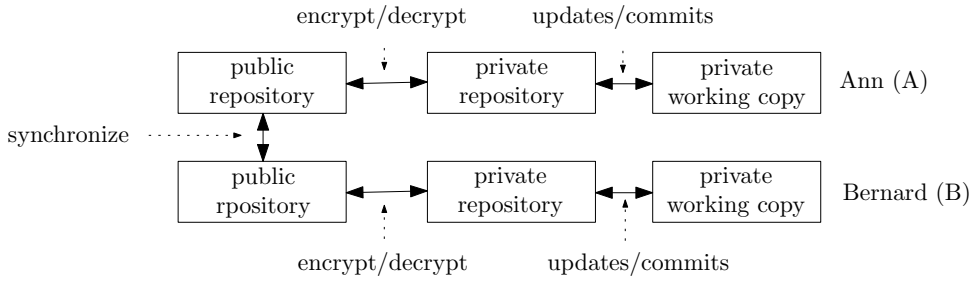

Figure 3. The repositories and working copies of two users.

data and all deltas from the clear repository. The deltas will of course be encrypted in this repository. All version control operations between working copy and clear repository work as usual (including rollbacks and conflict handling), since they are both on the same host. Synchronization between hosts is done by synchronizing the encrypted repositories in both hosts. The encrypted repository will be called the "public" one, and the clear repository the "private" one.

Figure 3 above shows user A (on top) and user B (at bottom). Each user has one private repository and working copy, and also one public repository. Signatures from private repositories are also encrypted, and decrypted back in the other repository; another signature may added to the encrypted delta.

In distributed version control, each user decides which other users he will trust (and consequently which deltas will be accepted when synchronizing repositories), so we can assume that each repository has one single administrator. Only the Grant and Revoke operations are defined:

A user A grants access to a user B on his repository by adding B's public key to his public repository, and encrypting the keymap $\mathcal{M}$ and the keys $\mathcal{K}$ with B's public key. When B synchronizes with A's public repository, he will be able to decrypt all deltas using his private key.

User A revokes access from user B on his repository by removing B's public key from his public repository and immediately picking a new symmetric key k. If B has access to A's repository, he will not be able to decrypt the new deltas that were encrypted with k.

For example, suppose A and B are using the same symmetric key, k. A decides to revoke Eve's access to the repository, and generates a new key k'. Meanwhile, B revokes Malice's key, and generates a new key k''. If A and B agree on honoring each other's revocations, then they may keep the intersection of authorized users only. After they synchronize their databases, a third key k''' (that neither Eve nor Malice knows) should be created. This situation can be detected by checking the keylists during synchronization of public repositories. Let $\mathcal{G}^A$ and $\mathcal{G}^B$ be two lists of users before the synchronization of the public repositories; $\mathcal{K}^A$ and $\mathcal{K}^B$ the two key lists. Then the new user list $\mathcal{G}$ and key list $\mathcal{K}$ can be generated as:

$\mathcal{G} \leftarrow \mathcal{G}^A \cap \mathcal{G}^B$
$\mathcal{K} \leftarrow \mathcal{K}^A \cap \mathcal{K}^B$
If $\mathcal{G}^A \not\subseteq \mathcal{G}^B$ and $\mathcal{G}^A \not\supseteq \mathcal{G}^B$, generate a new key k''' and $\mathcal{K} \leftarrow \mathcal{K} \cup \{k'''\}$

And k''' is used as the latest key.

3.3.1. Synchronizing private and public repositories

The encrypted deltas are stored in the public repository along with their interdependencies (the deltas and their interdependencies form a directed acyclic graph, as mentioned before, and the synchronization algorithm needs to list the deltas in topological order). The public and private repositories may be synchronized using Algorithm 1.

Input:
Δ^U, the set of all deltas in the public repository;
Δ^R, the set of all deltas in the private repository;
W^U, a scratch working copy for the public repository;
W^R, a scratch working copy for the private repository;
$\mathcal{K}$, the list of symmetric keys;
$\mathcal{R}$, the list of revisions;
$\mathcal{M}$, the mapping of keys onto revisions.

1 Checkout ($W^U, 0$);
2 Checkout ($W^R, 0$);
3 **forall** $\delta_{i,j} \in \Delta^R \setminus \Delta^U$ **do**
4 $\quad k \leftarrow \mathcal{K}_{\mathcal{M}(j)}$;
5 $\quad$ Update (W^U, i);
6 $\quad$ Add($W^U, \{\delta_{i,j}\}_k$);
7 $\quad$ Add($W^U, dep(i,j)$);
8 $\quad$ Checkin(W^U, j);
9 **end**
10 **forall** $\delta_{i,j} \in \Delta^U \setminus \Delta^R$, *in topological order* **do**
11 $\quad k \leftarrow \mathcal{K}_{\mathcal{M}(j)}$;
12 $\quad$ Update (W^R, i);
13 $\quad$ Apply ($W^R, \{\delta_{i,j}\}_k$);
14 $\quad$ Checkin(W^R, j);
15 **end**

Algorithm 1: Synchronization of public and private repositories.

In the synchronization algorithm, we use the following functions:

- **Checkout (**W, i**)**: checks out revision i into working copy W. In the case of the first private working copy, Checkout ($W^R, 0$) will use the first symmetric key from the list;
- **Apply (**$W, \delta_{i,j}$**)**: apply $\delta_{i,j}$ to W, so the working copy will be identical to revision j;
- **Add (**W^U, δ**)**: given a delta (δ), *store it as a file* in the public repository;
- **Add (**$W^U, dep(i,j)$**)**: in the public repository, add the edge (i, j) to the graph that represents delta interdependency;
- **Checkin (**W, i**)**: commits changes to the working copy W, using the revision number i;

Operation	Time (centralized)	Time (distributed)
Import	$s(\|\delta_0\|)$	—
Checkin	$a(\|k\|) + s(\|\delta_i\|)$	—
Checkout	$s(\|\text{content}\|)$	—
Update $v_a \rightarrow v_b$	$a(\|k\|) + \sum_{a<i<b} s(\|\delta_i\|)$	—
Grant access	$a(\|\mathcal{K}\| + \|\mathcal{M}\|)$	$a(\|\mathcal{K}\| + \|\mathcal{M}\|)$
Revoke access	$a(\|\mathcal{K}\| + \|\mathcal{M}\|)\ \|\mathcal{G}\|$	$a(\|\mathcal{K}\| + \|\mathcal{M}\|)\ \|\mathcal{G}\|$
Synchronize(r_1, r_2)	—	$\|\Delta^{r_1} \setminus \Delta^{r_2}\| + \|\Delta^{r_2} \setminus \Delta^{r_1}\|$

Table 1. The cost of different operations using the proposed protocols.

3.4. Performance and memory overhead

The proposed protocol is light – Table 1 shows the time required by each operation. In the table, k is the symmetric key, δ_i is the data that represents the difference between two versions of the content, and $|x|$ is used for "size of x". r_1 and r_2 are two repositories in a distributed system. $s(n)$ stands for "the time taken to encrypt or decrypt an object of size n with the symmetric key k"; $a(n)$ is " the time taken to encrypt or decrypt an object of size n with one of the asymmetric keys K_Z". The set of deltas in repository r is Δ^r. The operation "Update $v_a \rightarrow v_b$" is the update of a working copy from version v_a to version v_b. Note that for the checkout operation, the cost is proportional to the current size of the content. The only operation that is affected by the number of users is the revocation of a compromised key (because of the time it takes to encrypt the new key for all users); the only operation that is affected by the total size of the content is the initial import. All other operations depend on the size of deltas and on the size of the symmetric key (which is negligible).

When a user performs a checkout or update, only the relevant part of the key list is retrieved (that is, the symmetric key encrypted with his private key – the keys encrypted for other users need not be retrieved).

If the cryptossystem or PKI needs to be changed, it may be necessary to re-encrypt the whole key list history (which would not likely be too large anyway).

Both protocols require that each user stores additional information on his workstation. The volume of extra information required is the size of the public repository (this will depend on the compression algorithm used before the symmetric cipher and on the size of the repository); the size of the key list (which depends on how often the symmetric key is changed); and the keymap, which only adds one key ID and one delta ID per delta. There is also the size of the keyring in the PKI, which depends on the number of users.

4. The implementation

We have implemented the protocol for distributed systems. The prototype is called *Apso* and was written in C++. Figure 4 shows the core architecture: a PubRepository object has a Keystore, a CryptEngine and a VCRepository (an abstraction layer for version control systems). The CryptEngine and VCRepository classes should be extended to describe actual cryptographic libraries and version control systems (we have implemented a plugin for the Nettle cryptographic library [10] and a plugin for the Monotone version control system). Nettle was chosen because it is simple and small. Monotone was chosen because

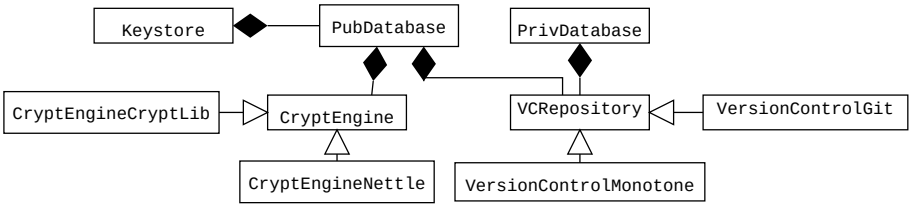

Figure 4. UML diagram for the core architecture of Apso.

it has direct support for extracting and injecting deltas. The Git and CryptLib plugins were not implemented, and are there only as illustrations. Besides the core architecture, there is a command-line interface and a Façade class for users who want to use Apso as a library.

Monotone is a totally distributed version control system that supports (and enforces by default) the signing of each commit made to the repositories, and uses a hash of the delta as revision number. Each user may decide on a list of other users whose keys will be accepted when incorporating changes to his local repository (which in Monotone jargon is a "database"). Nettle is a cryptographic library that is small and only performs cryptographic operations and does no resource allocation or I/O.

4.1. Implementation details

A public Apso repository is always linked to a private one and contains the following tree:

```
apso/deltas/
apso/users/USER/public_key
apso/users/USER/keys/
apso/users/USER/keymap/
```

Where `deltas` is a directory where all the encrypted deltas are stored (filenames are revision ids). Each user has one `USER` subdirectory with a file `public_key` containing that user's public key, and also two subdirectories: `keymap`, where each file maps one revision onto a symmetric key id (in version control systems which support versioning of links, these are links to actual keys), and `keys`, where all symmetric keys are stored (filenames are key ids). Users keep their private keys wherever they find convenient, and pass the key location to Apso .

For example, Alice has her key list under `apso/users/alice/keys`. When she needs a specific key k_i, she will find it in a file `apso/users/alice/keys/key_i`. Just as k is in the file `apso/users/alice/keys/k`, the mapping for revision r is in the file `apso/users/alice/keymap/r`. It may be good to deny access to Alice to other subdirectories, with other users' keys, but the key lists are encrypted so it may be unnecessary.

All version control operations work as usual on the private repository, and only these three operations are defined in our implementation:

- **Private and public repository synchronization**: this is done using algorithm 1;
- **Granting access**: the administrators will encrypt all the symmetric keys and check them in (in the public repository), so the user may check out his list later;
- **Revoking access**: someone in the admin group adds a new key to the list and removes the user's directory `apso/users/USER/` from the public repository.

Apso was tested withs AES as the symmetric cipher and RSA for encrypting the AES keys (both provided by Nettle).

5. Conclusion

This paper described protocols for bringing secrecy to concurrent revision control systems. The protocols do not require servers to ever see either the clear content or the key used to encrypt it. Keys are encrypted using asymmetric algorithms, but the content (which is supposedly larger than users' keys) is only encrypted using symmetric ciphers. The protocols also work for the non-merging revision control systems (those in which locks are used to avoid conflicts). We have implemented the distributed protocol using an extensible architecture. We plan to implement the same protocol in other modern version control systems and use Elliptic Curve Cryptography [5] to speed up the revoke operation when the number of users and the key history are large.

References

[1] Arch: a distributed version control system. `http://www.gnuarch.org/` (May, 2006).

[2] Darcs: a revision control system. `http://abridgegame.org/darcs/` (May, 2006).

[3] Jacky Estublier, David Leblang, Geoff Clemm, Reidar Conradi, Walter Tichy, André van der Hoek, and Darcy Wiborg-Weber. Impact of the research community on the field of software configuration management. *Transactions on Software Engineering and Methodology*, 14(4):1–48, 2005.

[4] Git: Tree history storage tool. `http://git.or.cz/` (May, 2006).

[5] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004.

[6] E. J. Whitehead Jr. and Annita P. Dahlqvist, editors. *Proceedings of the 12th International Workshop on Software Configuration Management (SCM 2005)*, 2005.

[7] Alexis Leon. *Software Configuration Management Handbook*. Artech House, 2 edition, 2004.

[8] J. MacDonald. Versioned file archiving, compression, and distribution. UC Berkeley. Available at `http://www.cs.berkeley.edu/~jmacd/` (May, 2006).

[9] Monotone: a distributed version control system. `http://www.venge.net/monotone` (May, 2006).

[10] Nettle: a low-level cryptographic library. `http://www.lysator.liu.se/nisse/~nettle/` (May, 2006).

[11] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion*. O'Reilly, 2004.

[12] Dave Thomas and Andy Hunt. *Pragmatic Version Control Using CVS*. Pragmatic Bookshelf, 2003.