Trustversion: A Secrecy-protected Version Control System

Marta Isabel Ribeiro Sequeira
Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal
Email: marta.sequeira@tecnico.ulisboa.pt

***Abstract*—Version Control Systems (VCSes) are widely used by software developers to store their software projects in remote locations and to maintain a record of all modifications in their files. Considering that software projects may constitute valuable private data for developers, having such projects stored in remote servers can raise security concerns as the servers may not be trusted. Currently, VCSes, in particular Subversion, do not provide data confidentiality in the face of compromised servers. In this project, we designed a system to protect the data called Trustversion, which provides a solution for performing SVN operations while keeping the developers' files encrypted on the Subversion repository. This solution protects the user's private data, even if they use online SVN repository services, such as SourceForge, hosted in remote servers that may not be trusted. Trustversion also ensures low storage usage overhead by taking advantage of the Subversion mechanisms for storing data efficiently.**

I. Introduction

Revision control has become an essential tool in project development for managing history of information and to allow multi-user contributions in documents. Examples of typical cases that benefit from revision control are software projects, LaTeX documents or system configuration files. Version Control Systems (VCS) implement this functionality by storing the files in another location usually remote, called repository, and providing methods for synchronization. The most currently popular VCSes are CVS [1] and Subversion [2], which follow a typical client-server architecture, and Git [3] and Mercurial [4], which adopt a peer-to-peer architecture. There is also now a variety of web services that provide online storage repositories powered by these VCS systems. Some examples of systems providing these services are: GitHub [5] supporting Subversion and Git, BitBucket [6] supporting Git and Mercurial, SourceForge [7] supporting Subversion, CVS, Git and Mercurial, and Google Code [8] and CodePlex [9], both supporting Subversion, Git and Mercurial.

Online repository hosting services are a convenient solution for users that do not possess the resources to create or manage remote repository servers. Furthermore, because these services guarantee high availability and some of them provide additional features to help with project management, such services are now widely used for storing revision controlled data, even for enterprise and personal usage. For example, SourceForge has currently over 430,000 software projects involving 3.7 million users [10] and GitHub has over 23.2 million projects involving 9.7 million users [11].

However, VCSes do not guaranty privacy of information on the server side. In fact, they are design to work solely with plaintext files. The only security guaranty these VCSes give is secure access control to files by users. This poses a threat to users entrusting private information to online servers susceptible to attacks such as intrusion or information disclosure. Let's imagine that a company uses a VCS hosting service to store a privacy-sensitive project that needs to be protected against the competition. The company will not be able to tell if the hosting server is compromised, either by an external attacker that was able to penetrate the server and has now access to all data stored in there, or a more simple internal attack by the service provider allowing information leakage. Even a curious or malicious system administrator constitutes a security concern.

Protecting the confidentiality of VCS files is not straightforward. The users or a simple system could try to protect the data by encrypting all the files before submitting them to the repository server, and decrypting upon receiving them from the server. However, VCSes employ mechanisms to avoid storing duplicated data which could be rendered ineffective. In fact, by "blindly" encrypting all file versions, VCS will not be able to detect which file parts remain immutable across versions and, therefore, it would store parts of the same data multiple times unnecessarily which would heavily increase the occupied storage on the server over time.

II. Goals and Requirements

This thesis aims to protect private data in public VCS services by securing the information on the repository through file encryption while allowing the VCS server-side operations to be performed over encrypted data. In order to achieve this main goal, our solution must fulfill the following set of requirements:

- *Data confidentiality*: Guarantee data confidentiality by cyphering all documents in the repository.
- *Storage space efficiency*: Consider storage space efficiency, since encrypted files can become much larger than plaintext files and the encryption schemes can

interfere with the VCS compression algorithms. Minimizing the overhead in storage space is particularly important in VCS service providers because it can influence pricing plans and increase costs for the users.
- *Client-side implementation*: The implementation of our solution must reside on the client side only. By avoiding alterations on the server-side code, our solution can be deployed independently of the server and could be used seamlessly with any existing online repository server.

To the best of our knowledge, there are currently no approaches that guarantee data confidentiality under this set of requirements. Recent work has shown to be possible to operate on encrypted data: to do arithmetic computations [12], [13], to search keywords [14], [15], [16], [17] and to sort the data [12]. Some systems were implemented or complemented with a combination of these techniques to reach the same goal of operating over encrypted private data, namely Database Management Systems and Private Information Retrieval systems. However, the approaches taken by these systems cannot be applied to securing VCS services because the operations involved in a Version Control System are a lot different from retrieving information or performing SQL queries with a fixed structure.

This thesis presents Trustversion, a system to ensure data privacy for public Subversion repositories. Trustversion provides confidentiality of private data while ensuring storage space efficiency, as it is coordinated with Subversion's mechanisms to reduce space usage. Our system runs exclusively at the client side and works with unmodified Subversion servers. An unmodified Subversion stores on the server only the changes made to the files and not every full version of the files. So the underlying idea of our solution is to encrypt those file changes that are sent to the server. Then we *trick* the server into believing that the user is just appending unrecognizable binary data to the files. The binary data is actually the encrypted file changes, but the server will be unaware of it. Because the encrypted information is being passed onto the server as append-only modifications on the files, Trustversion does not interfere with the normal behavior of Subversion. Moreover, by encrypting only the changes on the files, the encrypted information will be much more compact then a simple approach of encrypting the entire files. We call this method *append-only encryption*.

The implementation of Trustversion was made for Apache's Version Control System: Subversion [2]. We implemented an adaptation of Subversion supporting basic encryption of the data, simulating the approach of manually ciphering the data before sending it to the server. We also implemented another adaption of Subversion supporting append-only encryption, our solution that takes into account the Subversion's mechanisms for saving storage space by keeping file changes as encrypted appends on the original file. Both these systems were compared side-by-side to analyze the impact of interfering with Subversion's server mechanisms to effiently occupy disk usage.

The evaluation was performed on three different systems: the Subversion system, our implementation of a modified Subversion supporting basic encryption functionality and our solution, Subversion with append-only encryption. We applied these systems to four big software projects: CloudStack [18], CouchDB [19], Perl [20] and Ant [21]. We evaluated the results of these projects using the three systems in terms of storage space occupation and execution time performance, and compared them in order to see the impact of each approach and the overhead they introduced. To obtain the repositories to use as test cases and to be able to apply all three systems on them, we developed automated scripts to retrieve repositories from publicly available servers and to copy each repository by reproducing every revision in the repository and, thus, simulating the activity history of the original repository.

III. Background

The main application of VCSes has always been revision control which is the concept of registering changes in files with an associated revision number, which identifies each set of changes at a given point in time. A VCS always includes at least one repository which contains the tree of files that are stored in the VCS. The users participating in the same repository use it to synchronize changes and revisions through commands that are similar among all VCSes. Subversion's the most relevant commands are the following:

- the *checkout* to copy all the files in a repository to the user's local filesystem, creating a work area for the user to manipulate his files, called the *working copy*.
- the *commit* to associate a new revision number with the current set of file changes in the working copy since the last revision and submit them to the repository.
- the *update* to pull all changes submitted to the repository that have not yet been applied to the user's working copy.
- the *add*/*remove* to schedule files to be added or removed from the repository in the next commit operation.
- the *status* to print the current state of the working copy, namely if there are changes to be submitted.
- the *diff* to print textual differences between two specified revisions of the same files.
- the *log* to show the history of revisions and messages associated with them.

In this work we focus on the Subversion VCS, which follows a client-server model that includes one Subversion server and at least one Subversion client and libraries implementing the VCS commands to be used the clients. The client interacts exclusively with the *Client Library* through an API, which was designed to allow for various implementations of a Subversion client.

To understand how Subversion optimizes storage space usage, the internal behavior of the user operations on Subversion must be explained. When perfoming commits or updates Subversion uses two types of objects to perform remote function calls for transfering changes between the client and the server. The two types of objects are the *Editor* and the *Reporter*. The Editor applies the changes in the server or in the client depending on the operation. The remote function calls it supports are the following:

- *Delete Entry*: for deleting a directory or file;
- *Add Directory/File*: for adding a new directory or file;
- *Open Directory/File*: for opening a directory or file to make changes inside it;
- *Change Directory/File Property*: for altering the properties of a directory or file (for example, permissions or ownership of the directory or file);
- *Apply Text Delta*: for altering the contents of a file;
- *Close Directory/File*: for closing a directory or file after all changes have been made;
- *Close/Abort Edit*: to finish or abort (in case of error) the execution of this editor.

In the particular case of the commit operation, the remote functions are called by the client sequentially as a way to reproduce in the server the changes introduced in the working copy. In the case of the update operation, these functions are called by the server to update the working copy with the latest changes.

The report process provides the necessary information for the server to know which changes to apply on the client through the editor. The report process is performed by the client using a *Reporter* requested to the server.

The Reporter is how the server or clients report the current state of their repository or working copy, respectively. The report process provides the necessary information for the server to know which changes to apply on the client through the editor in the case of an update operation. The reporter has a much smaller set of functions, used only for informing which files are present in the repository/working copy and which revisions they are in.

Commands like *status* and *diff* behave in a way very similar to the update operation if the user specifies to also use information from a repository. They also use a reporter to provide information about the working copy and an editor to transmit changes. Although, unlike the update operation, these commands use editor objects that do not perform modifications on the files. The server calls the same editor functions as for an update, but they do not actually alter the state of the directory tree and rather just print information about the changes. As mentioned before, the *status* is for printing a description of changes in the directory tree, and *diff* is for printing differences in file contents.

The focus is on the commit and update commands because all other most commonly used Subversion commands perform the same overall procedures.

A. Managing File Content Differences

We've seen how changes in the tree of files are managed, in particular how to transmit changes in the content of a specific file. These changes are intended to save storage space on the server, since only the sections of the content that were modified are sent instead of the entire file. Subversion computes which parts of the file differ towards the version from last revision and generates a *delta* for that file. A delta is a description, with a compressed format, of the modifications introduced in a file and is generated by Subversion's diffing algorithm [22], [23]. This is why the function for altering files is called *Apply Text Delta*.

Deltas are how the server can store every version of every file in an efficient manner. The changes on a file in each revision are stored as a delta towards a previous revision of that file. The revision 0 is always an empty revision and the first commit of a file is stored on the server as a delta towards revision 0 (which translates to the entire file).

If we were to encrypt the entire files before committing them, the purpose of the deltas would be irrelevant. Performing a diff between two versions of the same file encrypted separately, produces two very distinct binary files, even if the plaintext versions are almost equivalent. A delta computed for these two versions of the file would probably be almost as big as the file itself, so it would not optimize storage space usage at all. Over time, the increase in storage occupied in the server would be proportional to the files sizes times the number of deltas for each file.

B. Managing Duplicated Information

Besides the delta mechanism described in Section III-A above, Subversion also has a mechanism to avoid storing duplicated information, called *Representation Sharing*. This mechanism optimizes storage space regarding *branches* and *tags*.

Both *branches* and *tags* are common concepts in VCSes. A *Branch* is a copy of the repository's current state which allows to perfom modifications to the repository's files in parallel with possible modifications on the original files. This is typically used in software projects to develop new (possibly experimental) features without interfering with current developments on the same project. All the modifications applied in a branch can then easily be discarded or merged with the current state of the original files (called *trunk* in Subversion). A *Tag* is a representation of a repository at a specific point in time. A tag saves the state of the repository in a certain revision and can add a meaningful name to it. In software projects, typically, each tag denotes a version of that software (e.g. "v1.2.1").

In the case of Subversion, the trunk, branches and tags are all simple directories in a repository. Each branch or tag is a complete copy of the tree of files and directories inside the trunk directory. This means that every time a branch is created, all the content in the trunk directory is duplicated.

Thus, in order to optimize storage space, Subversion keeps only one representation of the duplicated information on the server. If multiple unmodified copies of a file exist across the entire repository, only one file is kept on the server. Given that software projects widely use branches and tags, this can reduce a lot of overhead for big software projects.

IV. Related Work

There is current existing work among different areas that, similarly to Trustversion, address the general problem of securing privacy-sensitive user data and computations from a untrusted server. Although, these systems do not satisfy our requirements in securing remote VCS systems.

There is one system that tries to solve the problem of securing data in a VCS system, called Crypto Hook [24]. Crypto Hook is a TortoiseSVN [25] plugin, which is a graphical client implementation for Subversion. The main idea of Crypto Hook is to intercept the commit operation and encrypt all the files before they are committed to the server, and intercept the update operation to decrypt all files after the modifications are applied. More specifically, Crypto Hook ciphers every file in the working copy before the commit operation and then deciphers them so that the user can manipulate them. Crypto Hook also ciphers every file in the working copy before performing an update and deciphers them after, also decrypting the changes introduced by the update. Even though this plugin introduces a possible solution to the problem of protecting data in a Subversion repository, it has a few problems. Besides the system only working with a specific client implementation of the Subversion VCS, it does not take any advantage of the Subversion mechanism for storage efficiency. A Subversion repository only stores diffs representing the changes in the files in each revision, but a diff of an encrypted file in relation to its old version will be the new entire file, which will eventually generate a big storage expansion for that repository. This is not a desirable solution, taking into account that there are very large projects under revision control.

There are systems that also try to protect the user's private data in other areas such as Database Management Systems, Private Information Retrieval and systems with deduplication mechanisms. Even though several of these systems provide interesting ideas for dealing or directly manipulating ciphered data, none of them actually solves our problem of protecting VCS files.

V. Design

The goal of our solution is to maintain all file content encrypted while taking into account the diffing and storage mechanisms of Subversion to introduce minimum storage space overhead in the server.

Our idea follows the same reasoning as Log-Structured File Systems [26] and it is to store all changes on a file like *log-structured changes*, by treating every change as if it was only new data appended to the file. We call this process *append-only encryption* and it is represented in Figure 1 for a commit to revision 4. Imagine that Bob created the file `abc.txt` and committed the file to the repository for the first time. This generates revision 1, when the entire file is encrypted by Trustversion and sent to the server. Later, Bob can commit additional changes to the file, creating additional revisions. As explained in section III-A, when the client tries to commit a changed file, Subversion generates a *delta* with a description of the changes. In this case, Trustversion produces its own delta for append-only encryption. Our system obtains a description of the changes by performing a diff between the file and its previous revision, and then compresses and encrypts the diff. The result is then prepended with a string "TRUSTV" so append-only encryption can identify where a delta in a file begins. With append-only encryption, we append the Trustversion delta to the file's encrypted version that was committed in the previous revision and was encrypted using the same append-only approach. Therefore, we are placing the encrypted delta after a sequence of deltas already appended on previous revisions. After this process, Trustversion passes this new constructed file to Subversion. The reasoning behind adding the encrypted delta to the encrypted file from last revision and passing it to Subversion, as opposed to passing only the delta, is that Subversion will calculate on its own the file differences independently of Trustversion and generate its own delta before sending it to the server. Subversion keeps its own copy of each file from last revision, and uses it to compare with the corresponding file and generate the delta for commit. We want the delta produced by Subversion to indicate that the only changes in the file were the contents of our delta added to the end of the file from the last revision. In other words, we want Subversion's delta to contain only our delta. With this approach, we can make Subversion send only our delta as the only change performed on the file. By doing this for every commit, we trick the server into believing that all the changes are just new binary data (ciphered deltas) appended to binary files (ciphered files). We can see in Figure 1 that Bob committing a 4th revision

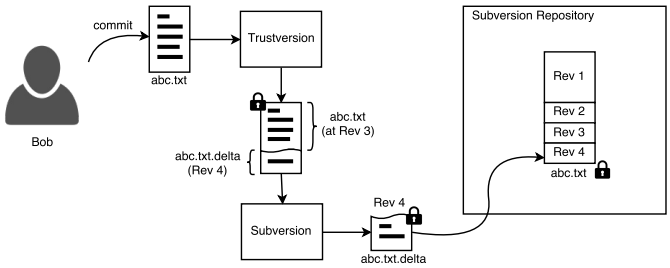

Figure 1. Management of file changes during a commit with Trustversion.

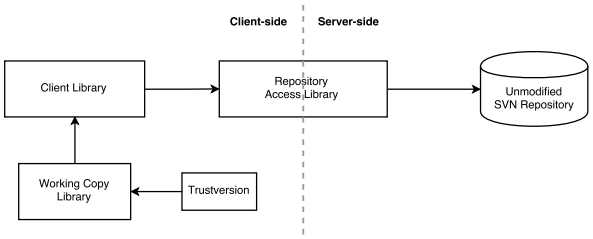

Figure 2. Architecture of the Trustversion system.

produces a 4th append on the `abc.txt` file from the point of view of the server.

On the other hand, when a client updates an out-of-date file, it will receive deltas from the server indicating only append modifications to the file. We, however, know that these modifications are actually Trustversion ciphered deltas containing the real modifications to the file. Knowing that the Trustversion deltas begin with the string "TRUSTV", to decipher the file using append-only encryption, we identify all the deltas by scanning the file for that string. From the beginning of the file until the first "TRUSTV" is found, the file contains its first version committed, also encrypted, so the first step is to decipher that part and use the resulting plaintext as the base for applying the changes in the subsequent deltas. In the situation presented in Figure 1, this would be the portion of the file `abc.txt` corresponding to revision 1. We then sequentially take each delta present in the rest of the file, in the order they appear, and we decipher it, we decompress it and we apply the differences it contains to the `abc.txt` file. We interatively construct the file since revision 1 until its latest revision. The file is completely decrypted when all the deltas are processed.

On the server the files are stored as pieces of sequential ciphered deltas and, because they are encapsulated in Subversion deltas, the solution does not interfere with the normal operation of the server. Furthermore, with this solution, the relevante diffing process to obtain the actual file differences is performed by our system in the client on the plaintext files, as opposed to diffing the already simply ciphered files which would cause the generated deltas to be almost as big as the entire ciphered file. And, as we already discussed, trying to diff two different files ciphered with regular encryption methods is rather useless for the purposes of saving storage space.

Figure 2 depicts the architecture of our solution, showing how Trustversion is inserted in the particular case of the Subversion system. It is clear that Trustversion operates only on the client side, working with the Subversion client without interfering with Subversion server's behavior.

Because Trustversion doesn't require the server's implementation to be altered, it benefits from the mechanisms already in place in the server. One important server mechanism is *Representation Sharing*, explained previously in Section III-B. For software project repositories with multiple branches or tags, this mechanism is crucial for saving storage space in the server as it remove as much duplicate information as possible. In order to ensure no impact in this server mechanism, Trustversion uses the same key to cipher all the files within a repository, so that adding full copies of the identical plaintext files produces identical encrypted files. Therefore, Trusversion allows the server to avoid duplicated information while guaranteeing data protection.

VI. Implementation

In this section we present two separate versions implemented on top of Subversion. The first version provides a basic ciphering functionality to Subversion using the same approach as Crypto Hook, described in section IV. The second version is the implementation of the append-only encryption system described in the section above.

A. Subversion with Basic Encryption

As discussed before in section IV, one possible approach for providing data security in a Subversion VCS is Crypto Hook [24]. It is equivalent to a repository user manually ciphering all his files before committing them to the repository.

We implemented our own prototype by modifying Subversion in order to see how this approach affects Subversion's mechanisms for saving storage space on the server when applied to large repositories. The prototype was implemented in C, directly modifying the Subversion's source code. We modified only the client-side of Subversion and did not alter the server's implementation. The encryption algorithm chosen for the files was the symmetrical encryption AES-256 in CBC mode, with a fixed IV and secret key for the entire repository.

The idea is simply to cipher every file in the repository before each commit or update and decipher the files after the commit or update finishes. In our implementation, we cipher only the files that Subversion actually intends to commit because the ciphering process runs only after Subversion determines which files were actually modified. Yet, it is still done before the diffing process is employed by Subversion, guaranteeing the deltas are generated only from ciphered data. These are regular Subversion deltas, produced by diffing two ciphered versions of the same files. Since the commit operation does not alter any files, we also keep a copy of each file before ciphering it in order to recover its plaintext version at the end of the commit as opposed to actually deciphering every committed file.

As for the update operation, we cipher every file in the repository before the update. The need for encrypting the files before the update is due to Subversion receiving deltas that were generated against the ciphered versions of the files

and applying these deltas directly on them. Therefore, they must already be ciphered when the deltas are applied in order for the update to function properly. Because the update operation typically alters files, we also have to decipher every file after the update.

This implementation does not alter in any way the Subversion's diffing mechanisms, it just simply encrypts the entire files every time an operation is performed on them. As we mentioned before, in this case, even a minor modification between two plaintext versions of a file will produce completely different encrypted versions of that same file. And being encrypted data, the deltas generated for this file can be as big as the encrypted file itself.

B. Append-only Encryption

With the implementation of our solution we took a different approach. We did not modify the Subversion code at all, and, instead, we implemented wrappers for the SVN operations. Although it removes some transparency for the user, that would have to use the Trustversion system explicitly, it can work independently of the version of Subversion the user has installed.

These wrappers are a set of Perl programs, each program performing one SVN operation, implementing the append-only encryption approach. The encryption algorithm used in this approach is the symmetrical encryption AES-256 in CBC mode, with a fixed IV and secret key for the entire repository. They fully support the most used SVN operations – commit, update, add, remove and status – and they work as follows. They perform append-only encryption operations on the repository files before and after they call the corresponding Subversion operation, delegating the actual behavior of the operation to Subversion.

As we have described in the section V, Trustversion generates its own deltas to append files before passing them to Subversion. In order to do this, we keep track of files' modifications, so, for that purpose, we maintain a copy of the plaintext and the ciphered versions of every file in the repository. A file's plaintext copy is the unciphered version of that file from the last time it was committed. The ciphered copy of a file is the ciphered version of that file, encrypted using append-only encryption, from the last time it was committed. Therefore, both copies of a file correspond to the exact same revision. We use the plaintext copies to identify which files have been modified by the user since the last commit, by comparing each one to its plaintext copy. We also use them to help build our append-only deltas by diffing the modified file against its plaintext copy. The ciphered copies helps us build new ciphered files on each commit, using append-only encryption, and to identify modified ciphered files after each update, in the same way we identify modified plaintext files. We also use the ciphered copies to help rebuild the corresponding plaintext versions after being modified in the update operation.

When performing a commit using Trustversion, we start by traversing the repository's directory structure and comparing every file to its plaintext copy to identify which files were modified. For every unmodified file we simply replace it by its ciphered copy, so that it also appears unmodified to Subversion when we invoke Subversion's commit. When a modified file is encountered, we create a Trustversion delta by performing a *diff* between the modified file and its plaintext copy, generating a file a description of the changes in the *diff format* [27]. We then compress the diff result using most typical Zip algorithm *Deflate*, and we cipher it using the AES-256-CBC symmetrical encryption. Finally, we insert the string `TRUSTV` at the beginning of the ciphered compressed diff resultng in a delta that we append to the ciphered copy of the file. As we have seen in section V, the string `TRUSTV` helps us later to identify the delta in the ciphered file. This describes the process of producing an encrypted file using the append-only method and can be seen in Figure 3. In the end, we make a new copy of the current plaintext version of the file before replacing it for the newly constructed encrypted file to pass to Subversion. Therefore, depending on whether the repository files have been modified since last revision or not, all of the repository files are, respectively, either encrypted with append-only encryption, or replaced by their ciphered copies that were encrypted with append-only encryption in previous revisions. After processing all the files, the Subversion's commit is invoked to perform the actual commit on this working copy. When Subversion's commit ends, we simply replace all the currently ciphered files by its plaintext copies.

Intuitively, performing an update operation using Trustversion is the opposite process. We also traverse the repository's directory structure, although in this case we start by replacing every file by its append-only encryption ciphered copy, and then invoke Subversion's update on the ciphered working copy. Only after Subversion's update finishes we traverse the repository with the purpose of finding which files were modified by the update. To achieve this we compare the currently ciphered files with its corresponding ciphered copies. Unmodified files are simply replaced by its plaintext copy, but modified files must be scanned for Trustversion deltas and processed in order to rebuild the plaintext files. Since we keep a copy of the ciphered and plaintext versions of each file on its last commit, we can use them to process only part of the encrypted file. We start by identifying where the new data in the encrypted file begins. We take the ciphered copy, since it contains the last ciphered version we have of the file, and use its file byte size as an offset for the updated ciphered file. Being an append-only approach, we know that all the new data must by exclusively at the end of file, so the part of the updated file from the beginning until that offset contains exactly the same content as the ciphered copy, and the new data added by the update operation must

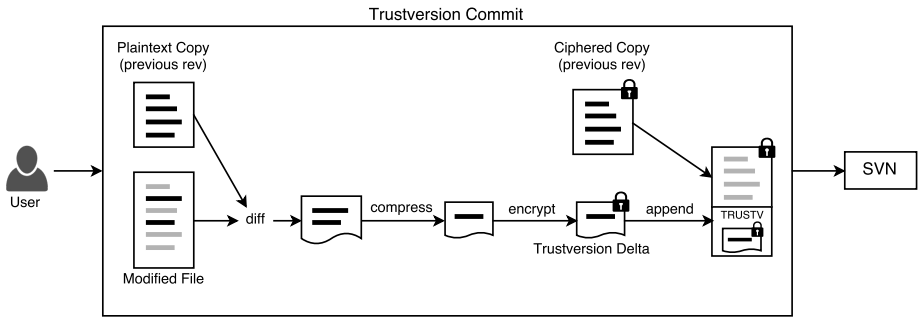

Figure 3. Implementation of the management of a modified file during a Trustversion commit.

start at that offset. So when we process the new data of the updated file, we scan for the string `TRUSTV` to capture all the deltas added in the update. Every time we find the string `TRUSTV`, we retrieve data from the file until the next `TRUSTV` appears or until the end of the file. These pieces of data are the Trustversion deltas. For each delta we retrieve from the updated file, we decipher it resulting in a compressed diff that we then decompress and use it to perform a *patch* of the changes on the plaintext copy. As we have said above, the plaintext and ciphered copies of a file correspond to the same revision. If we used the ciphered copy to process only the deltas added after the last revision, we can safely apply these changes to the plaintext copy from the same revision. Using this process, we can sequentially apply all the new Trustversion deltas in the file and avoid a complete rebuild of every file updated by Subversion.

VII. Evaluation

In this section we introduce the experimental evaluation to both approaches described in the previous section: Subversion with basic encryption and Subversion with append-only encryption. We present and compare its results to analyze the advantages of our solution. We also present the work involved in acquiring test cases and achieving these results.

A. Acquiring and Recreating Repositories

To really capture the impact Trustversion can have on a repository in terms of time and storage space overhead, we applied our system to repositories with a great number of files and/or revisions. We were also interested in practical and typical cases for repositories so we decided to use publicly accessible repositories of big software projects.

To apply both implementations on an already existing repository, we had to be able to recreate each revision since the beginning of the repository. In order to do this, we first implemented a repository "crawler" that would retrieve the entire revision history of public repositories. Then we needed to implement a repository "replayer" that could reproduce the entire history of a repository revision by revision.

The repository crawler is a script that tries to capture all the changes made in a repository since revision 0. This is only possible with publicly available repositories with unlimited read access, so we could have the script running autonomously, continuously invoking SVN commands on the repository.

The first step of this script is to invoke the `svn log` command on the repository, which gives us the list of all the revisions and its authors, dates and commit messages. We want to gather as much information as possible to be able to later use the replayer to recreate a new repository equivalent to the original repository. We then iteratively retrieve the repository's history. For each revision, we use the `svn update` and `svn diff` commands to download all files in the repository on that revision and obtain a list of which files were modified, added and removed. We store this along with the revision's number, author, date and commit messages that we gathered previously. This way we keep the complete repository state for every revision.

We decided that an efficient way to save this information would be in another Subversion repository. So the script creates an empty local repository and each commit to the repository contains a revision from the original repository, meaning the revision's files and the revision's metadata.

The importance of retrieving a repository's history and saving it locally is that it allows us to *replay* the original repository as many times as we want and in controlled environments where the results are not affected nor dependent of the accessibility or availability of a public remote server. In fact, while running the crawler with public repositories, we experienced problems regarding availability and connectivity to the servers. Some of the problems we faced included:

- Servers limiting the number of VCS operations per minute;
- Dropped connections during VCS operations involving large data transfers;
- Network connectivity problems;

- Network latency.

We adapted the script to circumvent some of the issues. For example, we added a delay between consecutive SVN command calls to avoid reaching the limits imposed by repository servers. We also modified the script to restart from the point where the connections were dropped. Dealing with these issues during the execution of the crawler, whose goal is to locally store the repository's history, allows us to later recreate repositories using our solution with more reliable results and more accurate measurements.

The phase for reproducing repositories is performed by the "replayer" script. As mentioned above, the repository replayer is a script capable of recreating an entire repository based on the information stored by the repository crawler. This script aims to generate a repository identical to its publicly accessible version at the point in time that the crawler was able to capture. We not only reproduce the file changes from each revision to the new repository, we also copy the authors and the commit messages. The goal of this script is to automatically interact with a repository while simulating the typical usage of a VCS with different users in real projects. We also want to gather data in order to analyze the impact of our solution in comparison to other versions of Subversion and obtain results mainly concerning the execution time of each VCS operation and the disk space occupied in each revision.

To replay the repository, we apply and commit the file changes revision by revision starting from the first revision, similarly to the crawler. As explained above, the chosen method for the crawler to store the information was a Subversion repository. This means that the replayer always requires a regular unmodified SVN installed to recover the information stored in the crawler's repository. So the replayer uses two versions of the Subversion system, the regular SVN to get the information saved by the crawler and a second version depending on the system we choose to replay the repository with. If we want to replay a repository using append-only encryption, the replayer uses both the regular SVN and the append-only encryption Subversion, where the latter is the system used to perform the VCS operations on the new repository that simulate the changes made in the original repository.

To replicate a revision, we start by using the regular SVN to perform an `svn update` and get the files from the crawler's repository on that particular revision. Besides the repository's files, we also get a list of which files were modified that contains the information about which of those files (or directories) were added or removed in this revision. Using the chosen SVN implementation, for example append-only encryption Subversion, we perform an add or remove operation on the corresponding files or directories. We also use the information about the author and commit message of the original revision and perform an append-only commit of the added, removed and modified files using the original commit message and pretending to be the original author. This way we get an identical revision on the new repository. We apply this process orderly for every revision of the original repository to replicate its entire history.

During the execution of this script, we also record some data regarding the time of execution of every VCS operation and the occupied space by the repository both in the client and the server, so we could process this data and analyze the impact of each system we implemented. The results are presented and discussed in the following section.

B. System Performance

We evaluated 4 big software projects with 3 different systems: the unmodified Subversion, the modified Subversion with basic encryption and Trustversion, our solution. Our goal was to obtain data using the regular Subversion system to serve as a baseline for comparison and obtain data using the other two systems to observe the overhead each system introduces by adding encryption on this VCS. It was also our goal to empirically show how our solution is preferable to simply ciphering all the files, the approach provided by the Subversion with basic encryption, described in section VI-A.

All the tests with the replayer script using the three systems were performed in VirtualBox virtual machines running the Debian Linux distribution Each virtual machine was configured with one single-core CPU and 2GB of RAM. Every test required 2 virtual machines, one serving as the SVN client and the other serving as the SVN server, and every set of client-server VMs ran exclusively one test at a time. A test comprises one complete repository replay using one specific system.

The software version of the unmodified Subversion system used, which is that same system our solution relies upon, is Subversion 1.8. The Subversion system with basic encryption is also based on the Subversion 1.8.

We replayed each repository three times, using the three systems enumerated earlier. During every repository replay, we collected data regarding the execution times of each VCS operation performed and the storage space usage on each revision. We applied these tests on the following software projects: CloudStack [18] with 415 revisions, CouchDB [19] with 4081 revisions, Perl [20] with 11628 revisions and Ant [21] with 12480 revisions.

The results for the Perl project are a good example of what happens with each system as the number of revisions increases. The Perl project has 11628 revisions, 3 times more revisions than CouchDB, and seeing the storage space occupied on the server with each system in Figure 5, it is clear how over time a system without the append-only encryption approach for storage efficiency affects the server. With the same number of revisions as CouchDB (4081 revisions), it is already visible in Figure 5 that Perl replayed

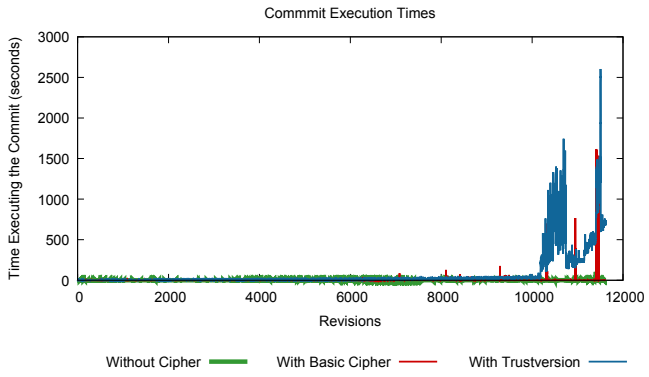

Figure 4. Execution time of every commit operation in the Perl repository.

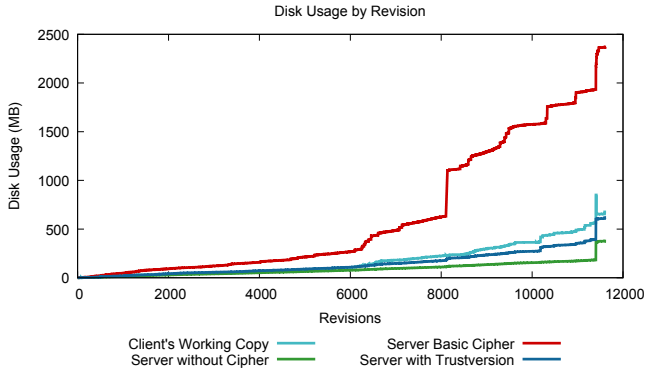

Figure 5. Storage space usage on the Perl repository.

with Trustversion is more efficient than using simple encryption. In fact, at that point in the repository's history, Trustversion has only 40.5% storage space overhead while Subversion with basic encryption has 219.5% storage space overhead, translating to 3 times more occupied space in the server than the unencrypted repository. But the significant evidence of Trustversion's ability to keep files encrypted and still be efficient with storage usage starts around revision 6000. Between revisions 6091 and 7136, most revisions were mainly constituted of modifications on existing files, and some of the commits performed included an average of 1000 modified files per commit. In that interval, there is an approx. 260MB increase in occupied space with simple encryption, whereas with Trustversion there is only an approx. 40MB increase. This is corroborated by the disk usage graph that shows us a clear impact on the server's disk storage when using Subversion with basic encryption and no visibly impact when using Trustversion. An even more accentuated impact of the same situation is visible between revisions 8105 and 8138, where each one of those 33 revisions is a commit of modifications to approx. 1200 existing files. With just 33 consecutive commits of modifications only, the server's occupied space jumped 470.08MB using Subversion with basic encryption while the Trustversion system caused an increase of only 18.78MB in the server. This is the type of situation where using Trustversion for data security is the most beneficial. When a revision or set of revisions are mostly constituted of modifications on existing files rather then added files, Trustversion is much more efficient than a simple approach for encrypting the files because, as we explained in detail in section V, the resulting revision *delta* of a modified file using Trustversion contains only the actual changes to the plaintext file, as opposed to a system with basic encryption whose resulting *delta* is the entire encrypted file. Multiplying this effect by 1200 files on 33 consecutive revisions creates the huge impact on the server that is so clearly visible in figure 5. By the end of the repository, Trustversion is able to achieve an overhead in storage space of 63.79% in relation to an unencrypted repository while Subversion with basic encryption reaches an overhead of 527.96%, corresponding to approx. 3 times more space occupied in the server than when using Trustversion.

Unfortunately, Trustversion's performance regarding the execution time of the commit operations suffers with the increase of files and repository size. At the end of the repository, the Perl project contains 70300 files with an average size of 8.95 bytes per file. This is mostly why in the last revisions of the repository, Perl's execution times, shown in figure 4, are not so great. Although, until revision 10185, Trustversion performs commits with an average execution time of 12.18 seconds. If we recall the discussion about Perl's disk usage graph, most of the revisions are primarily comprised of modifications on existing files. Therefore, the number of files in the repository did not change much during these revisions, and neither did the repository's size as we can see in that same graph, figure 5, so the execution times were not affected by these changes. But towards the end of Perl's repository, starting from revision 10185, a great number of files were added and the great majority of the add operations were *tags* and *branches*. As we have also explained before in section III-B, adding tags and branches can introduce a huge increase in number of files in the repository with minimal impact in the server's storage space. This is the reason why we only see a small storage usage increase in the disk usage graph in the revisions following revision 10185, even though those revision added up to 7736 files to the repository. The increase in repository files is what caused the spikes we see in figure 4. The decrease in execution times we also see around revision 11000 is due to some tags being removed from the repository, decreasing the number of files to be processed by Trustversion.

Overall, we consider that waiting an average of 76,68 seconds for a Trustversion commit for 1.7GB in saved storage space in the server is a reasonable tradeoff. Nevertheless, we believe that there is space for improvement on the append-only encryption system, in order to greatly improve these perfomance results.

VIII. Conclusion

Putting a software project under revision control typically means trusting a remote server with our private data, our software project files. However, we can't always trust a remote server. To address this problem, we propose Trustversion, a modified Subversion client that aims to provide a secrecy protected Version Control System. Our solution keeps all files encrypted in the remote server while avoiding a big increase in the size of the repository and also ensuring compatibility with any regular unmodified Subversion server. This is due to taking into account the current behavior of Subversion to store the encrypted files in an efficient manner, by introducing the idea of append-only encryption.

This system was tested and evaluated with big, well known software projects already under Subversion's revision control in order to gather quantitative measurements of its performance and resource usage overhead in comparison to both the unmodified Subversion and a Subversion with basic encryption functionality. We showed that Trustversion can indeed meet our goals and requirements for the solution, introducing significantly less storage space overhead than simply encrypting all files in the repository, without altering the server's implementation.

References

[1] "Cvs: Concurrent versions system," 2006, http://www.nongnu.org/cvs/.

[2] "Apache subversion. enterprise class centralized version control for the masses," https://subversion.apache.org/.

[3] "Git. distributed even if your workflow isn't." http://git-scm.com/.

[4] "Mercurial," http://mercurial.selenic.com/.

[5] "Github. build software better together." https://github.com/.

[6] "Bitbucket. unlimited private code repositories." https://bitbucket.org/.

[7] "Sourceforge. find create and publish open source software for free." http://sourceforge.net/.

[8] "Google code project hosting," https://code.google.com/.

[9] "Codeplex. project hosting for open source software." http://www.codeplex.com/.

[10] "Sourceforge about," http://sourceforge.net/about.

[11] "Github press," https://github.com/about/press.

[12] N. Z. Raluca Ada Popa, Catherine M. S. Redfield and H. Balakrishnan, "Cryptdb: Protecting confidentiality with encrypted query processing," *Proceedings of the 23rd SOSP*, pp. 85–100, Oct 2011.

[13] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," *Proceedings of the VLDB Endowment*, pp. 289–300, Mar 2013.

[14] S. Artzi, A. Kiezun, C. Newport, and D. Schultz, "Encrypted keyword search in a distributed storage system," *CSAIL Technical Reports*, Feb 2006.

[15] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan, "Building web applications on top of encrypted data using mylar," *NSDI'14 Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pp. 157–172, 2014.

[16] M. Bellare, S. Keelveedhi, and T. Ristenpart, "Dupless: Server-aided encryption for deduplicated storage," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 179–194.

[17] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, ser. SP '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 44–.

[18] "Apache cloudstack," 2016, https://cloudstack.apache.org/.

[19] "Apache's couchdb," 2016, http://couchdb.apache.org/.

[20] "The perl programming language," 2016, https://www.perl.org/.

[21] "Apache ant," 2016, http://ant.apache.org/.

[22] J. J. Hunt, K.-P. Vo, and W. F. Tichy, "An empirical study of delta algorithms," in *Proceedings of the SCM-6 Workshop on System Configuration Management*. Springer-Verlag, 1996, pp. 49–66.

[23] "Subversion notes: svndiff format," https://svn.apache.org/repos/asf/subversion/trunk/notes/svndiff.

[24] "Crypto hook," 2013, http://sourceforge.net/projects/cryptohook/.

[25] "Tortoisesvn: the coolest interface to (sub)version control," http://tortoisesvn.net/.

[26] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Log-structured File Systems*. Arpaci-Dusseau Books, 2014.

[27] "Diff unified format," http://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html.