http://doi.acm.org/10.1145/2462410.2462420

A Storage-Efficient Cryptography-Based Access Control Solution for Subversion

Dominik Leibenger
University of Paderborn
Warburger Straße 100
33098 Paderborn, Germany
dominik.leibenger@uni-paderborn.de

Christoph Sorge
University of Paderborn
Warburger Straße 100
33098 Paderborn, Germany
christoph.sorge@uni-paderborn.de

ABSTRACT

Version control systems are widely used in software development and document management. Unfortunately, versioning confidential files is not normally supported: Existing solutions encrypt the transport channel, but store data in plaintext within a repository. We come up with an access control solution that allows secure versioning of confidential files even in the presence of a malicious server administrator. Using *convergent encryption* as a building block, we enable space-efficient storage of version histories despite secure encryption. We describe an implementation of our concept for the *Subversion (SVN)* system, and evaluate storage efficiency and runtime of this implementation. Our implementation is compatible with existing SVN versions without requiring changes to the storage backend.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; E.3 [**Data Encryption**]

General Terms

Security, Design, Algorithms, Experimentation, Performance

Keywords

Access control; version control systems; subversion; convergent encryption; confidentiality; efficient storage

1. INTRODUCTION

Version control systems (VCS) have become an invaluable tool for software development and are also used during the creation of all kinds of documents. As their main feature, they allow restoring any previous version of a file. Most VCS also facilitate the collaboration of authors working together on a project. Authors obtain the latest copy from the VCS, work on that copy, and then commit an updated version to the VCS. A VCS will not normally store complete copies of each version of a file: changes might only affect a small fraction of that file, so only differences between two versions are stored.

One of the most common VCS is Subversion (SVN), which relies on a central server to store the history of versioned files. Like other VCS, SVN is not well-suited for working with confidential files. The transport channel is usually protected, e.g. using TLS. However, once on the server, each user with read access to the file system, as well as the server's administrators, can read the file. SVN allows to define access rules, but their enforcement requires a completely trustworthy server. Users can encrypt the content using external applications; unfortunately, this makes it impossible for the VCS to save only space-saving differences between file versions—even with only one byte changed in a long document, the encrypted versions will be completely different. Moreover, key management does not tie in with the versioning system. Even assuming there was a public-key infrastructure (PKI) in place, granting users read access to a file (and revoking that right later on) is a complex and error-prone task. If a user is supposed to get access to more than one file version, access rights must be granted (e.g., the file must be encrypted with the respective user's public key) each time a new version is uploaded to the SVN repository.

Our *contribution* is to define a solution for securing access to files in an SVN repository with the following properties:

- Rights can be managed individually for each file.
- Enforcing read access control through encryption allows protecting the confidentiality of files even against attacks by repository administrators.
- Despite encryption, space-saving differences are stored instead of entire copies of each file version. To achieve this, we allow attackers to see the positions of changes made between different versions of the same file.
- We retain full compatibility with old software versions, though not all features will work if either the client or the server does not support our solution.
- Necessary key exchanges tie in with the SVN architecture. They are authenticated without use of a PKI; a shared secret between a pair of users is sufficient.

The remainder of this paper is structured as follows: We discuss related work in Sec. 2 and outline our goals in Sec. 3. Sec. 4 presents the system design. The encryption scheme that allows for storage efficiency is worked out in Sec. 5; a security analysis follows in Sec. 6. Sec. 7 describes the implementation, which we analyze concerning storage / runtime overhead in Appendix A. Sec. 8 concludes this paper.

SACMAT'13, June 12–14, 2013, Amsterdam, The Netherlands.

2. PRELIMINARIES & RELATED WORK

While the concepts developed in this work might be applied to other VCS, we develop our access control solution for the SVN [19] system. SVN uses a client/server architecture. Each client stores a working copy, which contains both the versions of the files last obtained from the server and the versions edited by the user. After performing changes to files, the client performs a *commit* operation to add the changes to the repository; an *update* gets the latest contents from the repository. Communication with the server can be based on WebDAV [7], or can use a specific SVN protocol [4]. In both cases, the client only sends (or receives) the changes with respect to the previously stored version.

The server stores the whole revision history of the files and folders added to the repository as well as of property lists containing metadata that can be associated to them. After storing a revision, a user cannot change or delete contents of already stored data. Several storage backends exist. The most common one is FSFS [15, 3]; it stores the oldest version of each file completely. Once a file is changed, FSFS only stores the (binary) difference compared to *a* (not necessarily *the latest*) previous version. This way, SVN provides data deduplication between different revisions of individual files. Data deduplication between different files is not provided.

Our goal to develop a secure and storage-efficient access control solution for this system comprises two main challenges: A solution to allow convenient access rights management that includes key distribution, and a concept for data deduplication despite secure encryption. Work on secure file systems faces similar challenges; however, the majority of existing work focusses on the key distribution problem.

The secure file system *SiRiUS* [10] supports per-file access right management by encrypting each file using a randomly-generated key that is encrypted with the authorized users' keys and stored as part of the file's metadata. It supports separate read / write access rights.

Plutus [12] also distinguishes between read and write access rights, but groups files with equal access rights into *file groups* and stores their keys in an encrypted, shared *lockbox* to reduce key distribution costs. The authors introduce a method called *key rotation*: On access rights changes, keys are changed in a way that allows the computation of previous keys from the latest one, which prevents the need for immediate re-encryption of file contents on key changes. *Key regression* [9] fixes some security flaws of *key rotation*.

In contrast, in the *Wuala*[1] distributed file system, Grolimund et al. [11] introduce a data structure called *Cryptree* to explicitly store dependencies between *arbitrary* keys.

We use a simple, SiRiUS-like approach for key distribution as it fits best into the SVN architecture: The other solutions depend on relations between different files which are not guaranteed to be available in SVN setups (for example, a user might check out only a single file or directory).

Despite those systems that focus on key management, few secure file systems provide data deduplication.

Farsite introduces a concept called *convergent encryption (CE)* which we adapt in Section 5: They use a cryptographic hash of a file as a key to encrypt that file, so identical files can be recognized and deduplicated despite secure encryption. Similar to the SiRiUS approach, that key is then encrypted with the authorized users' keys to grant them access rights.

The concept has been extended in [18]: The authors suggest applying CE to chunks of a file, which are determined using a context-sensitive chunking procedure, to allow data deduplication even between similar files. Rashid et al. [17] recently re-used this concept to build a privacy-preserving data deduplication framework for cloud environments. We describe and further extend the procedure of [18] in Sec. 5.

The Tahoe filesystem [20], in contrast, uses CE for deduplication of entire files only. The authors introduce a *convergence secret*, so the equality of files can only be detected by users sharing that secret. This concept is similar to the *obfuscator* we introduce to hide equality of chunks between different files.

None of these approaches have been applied to VCSs. So far, besides using secure channels for communication, existing VCSs only cope with integrity and authenticity. Git[2] and Monotone[3] ensure integrity of the version history (if a trustworthy copy of the current version is available) and integrate the use of signatures. To the best of our knowledge, we are the first to provide a fine-grained, storage-efficient access control solution for a VCS that is resistant to a server compromise and thus able to provide *true* confidentiality.

3. GOALS, THREATS AND ASSUMPTIONS

The *goal* of this work is to protect the *confidentiality*, *integrity* and *authenticity* of specific file versions stored in an SVN repository by providing a fine-grained, user-definable access control solution. In more detail, our goals are:

- *Compatibility:* We limit ourselves to changes that only extend the SVN architecture, without restricting interoperability with existing SVN versions. This especially requires independence of our solution from the used storage backend.
- *Fine-Grained Access Control and Convenient Usability:* Our solution shall allow users of an SVN repository to protect their own files stored in the repository by defining access rights for them. Users shall be able to individually grant read and write permissions for a specific file, as well as permissions to manage the file's access rights, to other specific users. Access rights management shall be convenient insofar as key exchanges between users shall be performed automatically when access rights are managed (we only require a one-time manual key exchange per user for authentication).
- *Integrity and Authenticity:* If access rights have been associated to a file, our solution shall protect the file's integrity and authenticity: every authorized user shall be able to verify if the file's contents have been written by an authorized user according to the currently granted access rights on the file. This shall hold even against active attackers with full read/write access to the repository server. Note that since the SVN architecture expects only the server to store the whole revision history, we cannot provide protection against malicious deletion (including rollbacks that correspond to deleting revisions) by the server administrator.
- *Confidentiality and Storage Efficiency:* We want to provide confidentiality of files whose access rights have been restricted. Thus, attackers with full read/write access to the repository server shall not be able to read those files' contents. This shall be ensured through encryption—using an encryption scheme that is believed to provide IND-CCA-secure encryption.

[1] `http://www.wuala.com/`

[2] `http://git-scm.com/about`

[3] `http://www.monotone.ca/`

However, we want to retain storage efficiency, which SVN achieves by deduplicating data between different file revisions. As SVN's storage backend is transparent to the clients, data deduplication has to be done at the server side, i.e. without knowledge of file contents. This leads to a clear conflict with the confidentiality goal. We resolve this conflict by allowing to soften the security requirements when committing further revisions of a specific file. Allowing attackers to see the extent and positions of changes to a file (which includes information about how file contents are moved between revisions) enables generation of ciphertexts suitable for SVN's server-side data deduplication techniques. Users may decide for each file revision if this information leakage is acceptable. For files whose access rights change frequently, users may even decide to allow data deduplication between file revisions with different access rights—again by softening the security requirements. In this case, some additional information is leaked only to users who have or had legitimate rights to read some revision of a file. They are able to decrypt any parts of other revisions of the same file which were contained in an already-known version and might also be able to verify the existence of single plaintext fragments. Further, they are able to derive very limited information from chunk boundaries of other revisions of that file (cf. Section 5).
Note that this information leakage is limited to information leaked between revisions of a single file. We always hide similarities between different *files* as SVN does not perform data deduplication between files anyway.

We achieve these goals under some *assumptions*: As authentication is password-based, we assume that attackers do not know (and cannot break) these passwords. We further assume that no attacker can break the Diffie-Hellman key exchange, and that used hash functions provide resistance against preimage attacks as well as strong collision resistance. Moreover, we assume the existence of a pseudorandom permutation (PRP) which we use as encryption primitive. Users are considered authorized for specific actions w.r.t. a file version if they are granted the corresponding access rights by a user with GRANT rights (i.e. a user allowed to manage access rights) for that file version, as described in Section 4.1. A file owner and other GRANT-authorized users (as well as their SVN clients) are considered trustworthy w.r.t. the specific file.

4. SYSTEM DESIGN

Our access control solution consists of two building blocks: First, we design and integrate a suitable access control concept for the SVN architecture. Second, we adapt an encryption scheme that achieves confidentiality without preventing the use of data deduplication techniques such as SVN's difference calculations. This section deals with the former.

4.1 Overview

The idea of our access control solution is to let the *owner* of a file (i.e. the user adding that file to the SVN) decide on its precise access rights. The following rights can be granted:

- *READ:* Allow to read the contents of the specific file.
- *WRITE:* Allow to modify a specific file's contents.
- *GRANT:* Allow to grant arbitrary rights to and revoke arbitrary rights from other users.

To this point, this definition of access rights seems rather straightforward. However, the semantics of access rights granted for a specific file are not that clear when rights can be changed over time. While *WRITE* and *GRANT* rights can be *granted to* as well as *revoked from* a user effectively at any time, the impact of a *READ* right revocation is less certain: The user could have made copies of the file revisions he had access to, thus revoking the *READ* right would only be effective regarding future but not previous revisions. To make the *effective* rights clear and comprehensible, we tie our access control mechanism to single file *revisions* instead of whole files. In particular, we store all access-rights-related information as metadata for a specific file revision. This leads to minor limitations: According to the revision concept of the SVN architecture, granted rights of archived file revisions are immutable—a user is only able to change these rights for future revisions of the file (so each access right modification requires a commit). However, albeit rights are tied to revisions, modifying a file or its access rights requires the *WRITE* or *GRANT* right for the *latest* revision, thus these rights can be revoked with immediate effect. Solely a *READ* right for an old revision can be exercised by a user even if he does not hold the right for the latest revision.

4.2 Access Rights

As described before, we distinguish between the rights *READ*, *WRITE* and *GRANT*. They are enforced as follows:

- The *READ* right is enforced by encrypting the file contents using a key triple (K_R, K_O, K_I) (for details see Sec. 5) that is generated randomly by the file owner or any *GRANT*-authorized user and only made available to *READ*-authorized users. All cryptographic operations are performed at the client side and the keys are unknown to the repository, so the server can only see the ciphertexts. Thus, access control regarding this right can be resistant to a server compromise. While the repository does not know the file contents, server-side efficient storage of similar file revisions is retained by using a special encryption scheme that is described in Sec. 5.
- *GRANT* access control is performed using a shared secret key K_G (again generated randomly by a *GRANT*-authorized user like the file owner) that is only known to *GRANT*-authorized users. This key is used to prove / verify the integrity / authenticity of access rights, trust relationships and cryptographic keys.
- The *WRITE* right is enforced by a simple server-side ACL, since the repository administrator can modify or delete ciphertexts anyway due to his physical access to the storage.

Whenever access rights are changed, the affected keys are replaced by new keys that are chosen randomly by the *GRANT*-authorized user's client that performs the change.

4.3 Authentication and Key Exchange

To allow a convenient exertion of access rights, we do not require the users to exchange those cryptographic keys manually. Instead, every user u has to remember a single password $CP(u)$ (which *must* differ from the one used to authenticate to the repository) that allows him to exercise all of his access rights. We use the repository as transport channel for communication between users, so we need to minimize the number of communication rounds. This works as follows:

1. An individual key $K_{DH}(f, u)$ is negotiated for every user u of a file f using a Diffie-Hellman key exchange [6]. This key is made available both to the specific user u as well as to every *GRANT*-authorized user of file f. For this, we

regard the group of $GRANT$-authorized users of a specific file f as one party of the DH key exchange and the user u as the other party. The private DH value of the $GRANT$ group is chosen by any $GRANT$-authorized user (e.g. the file owner); the private DH value of u is chosen by u. We store the public DH values in the file's metadata. The private values of the $GRANT$ group and of u are stored encrypted with K_G and $CP(u)$, respectively, so that both u and $GRANT$-authorized users are able to determine $K_{DH}(f, u)$ later on.

2. As the DH key exchange does not provide authentication, we authenticate the negotiated DH keys separately. The idea is to build bidirectional trust chains from the owner of file f (who is assumed to be trusted and serves as trust anchor) to every user u—i.e. every user u is authenticated by the file owner over a chain of trusted $GRANT$-authorized users and vice versa. For this, every user u negotiates a secret passphrase with an arbitrary $GRANT$-authorized user using a separate channel (e.g. telephone), which is used by them to authenticate each other. We use a symmetric HMAC-based authentication mechanism; details of this protocol are omitted due to space restrictions.

3. The access rights granted to a user u are stored in the file's metadata and authenticated using an HMAC keyed by K_G so that modifications can be detected by $GRANT$-authorized users. To allow exertion of access rights, the keys associated to an access right are stored within the file's metadata—encrypted with the key $K_{DH}(f, u)$ for every authorized user u. Since $K_{DH}(f, u)$ is known to all $GRANT$-authorized users, changed keys can be distributed easily to the users by simply encrypting the new keys with $K_{DH}(f, u)$.

5. ENCRYPTION SCHEME

In principle, we could use any secure encryption scheme for encrypting file contents. However, traditional schemes would prevent data deduplication between similar file revisions. To achieve storage efficiency, we use a special encryption scheme that we describe in this section.

Unfortunately, data deduplication and secure encryption are conflicting goals as the former has to depend on similarities between contents which the latter requires to hide. This leads to an inevitable trade-off. The benefit of our encryption scheme lies in the fact that it allows to adjust this trade-off: The encryption of single file revisions achieves the same security properties as the underlying encryption function (e.g. AES). However, when further revisions are stored, the user may choose to leak a limited amount of information to allow data deduplication.

Towards this goal, our encryption scheme requires a triple of keys (K_R, K_O, K_I) instead of a single encryption key. K_R corresponds to the classic encryption key that is used to ensure confidentiality. It may be used for an arbitrary amount of encryptions, but has to be changed whenever access rights of the encrypted content change. K_I is used similarly to ensure integrity / authenticity. K_O—the *obfuscator*—serves as a kind of *secret* initialization vector (IV). Using a unique IV for every encryption achieves the strongest security guarantees. However, in contrast to other schemes, using the same IV for multiple file revisions is secure insofar as it only leaks a controlled amount of information: Fixing K_O only results in deterministic encryption—both regarding a whole plaintext as well as variable-length chunks within a plaintext. By revealing which parts have changed between two revisions, this allows data deduplication between ciphertexts of similar file revisions. Thus, a user may achieve storage efficiency by re-using K_O of an ancestor when creating a new file revision.

To achieve this, we adapt the concept proposed by Storer et al. [18]. Their idea is to split a plaintext m into so-called *chunks* using a context-sensitive chunking procedure before encrypting each chunk deterministically using convergent encryption (see Section 2). This way, identical chunks result in identical ciphertexts. The concatenation of those encrypted chunks forms the ciphertext of m. Unfortunately, this solution would have some shortcomings in our usage scenario: First of all, it would not only leak similarities between different *revisions* of a file, but also between different *files*, as contents are encrypted independent of the encryption key. Furthermore, due to the context-sensitive chunking, chunk boundaries leak some information about the plaintext m. Even worse, deterministic encryption of chunks could also leak information about the structure of m, as repeating contents would lead to repeating chunks and thus to repetitions in the ciphertext.

To overcome these shortcomings, we modify their concept in two points: First, we make the chunking procedure and the computation of chunk-specific keys depend on K_O to limit CE's security implications to the scope of a specific value of K_O. Secondly, we introduce *appearance counters* which are involved in chunk boundary determination and chunk encryption to prevent potential security risks regarding files with repeating contents.

We continue with a formal description of our scheme.

5.1 Formal Description

In this section, we describe our encryption and authentication scheme $\Pi = (\textsc{Gen}, \textsc{EncMac}, \textsc{Dec})$ following the notations defined in [13]. We later analyze our encryption scheme's security in the random oracle model [1], so we assume the existence of a random oracle H. As we focus on the encryption scheme, we refer to the contents of files and file revisions as *messages* in the remainder.

Requirements

We build upon an existing *pseudorandom permutation (PRP)* F (with $F_k(x)$ denoting $F(k, x)$) as defined in [13, Section 3.6.3] analogous to [13, Definition 3.23]:

Definition. *Let $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ be an efficient, length-preserving, keyed function. We say that F is a* pseudorandom permutation *if for all probabilistic polynomial-time distinguishers D, there exists a negligible function* negl *such that: $\left|\Pr[D^{F_k(\cdot)}(1^n) = 1] - \Pr[D^{f(\cdot)}(1^n) = 1]\right| \leq \mathsf{negl}(n)$, where $k \leftarrow \{0,1\}^n$ is chosen uniformly at random and f is chosen uniformly at random from the set of permutations mapping n-bit strings to n-bit strings.*

Further, we build upon an existing MAC function $\Pi_M = (\textsc{Gen}_M, \textsc{Mac}, \textsc{Vrfy})$ with unique tags. Again, we expect $\textsc{Gen}_M$ to return a uniform random number in $\{0,1\}^n$.

Utilizing the random oracle H, we define two different hash functions $\mathrm{H}_R, \mathrm{H}_S : \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^* \to \{0,1\}^n$.

Definition of Π

We construct Π from a to-be-defined private-key encryption scheme $\widehat{\Pi} = (\widehat{\textsc{Gen}}, \widehat{\textsc{Enc}}, \widehat{\textsc{Dec}})$ and the MAC function Π_M according to [13, Construction 4.19]:

- $\textsc{Gen}$, on input 1^n, runs $\widehat{\textsc{Gen}}$ and $\textsc{Gen}_M$ to obtain keys K and K_I, respectively.

- EncMac, on input key (K, K_I) and message m, computes $c \leftarrow \widehat{\text{Enc}}_K(m), t \leftarrow \text{Mac}_{K_I}(c)$ and outputs (c, t).
- Dec, on input a key (K, K_I) and a ciphertext (c, t), outputs $\widehat{\text{Dec}}_K(c)$ if $\text{Vrfy}_{K_I}(c, t) = 1$; otherwise, $\perp$ is returned.

Definition of $\widehat{\Pi}$

The key-generation function $\widehat{\text{Gen}}$ is defined as follows: On input 1^n, it outputs a key tuple (K_R, K_O) with K_R, K_O each being chosen uniformly at random from $\{0,1\}^n$.

The encryption function $\widehat{\text{Enc}}$ is more complex: When getting a key $K = (K_R, K_O)$ and a message m as input, it first splits the message into chunks, before performing separate encryptions for the individual chunks. The phases of $\widehat{\text{Enc}}$ are now described in detail.

5.1.1 Chunking

Chunking is performed in a similar way as introduced in [14] and presented in the convergent encryption scenario in [18]: A sliding window (fixed size: w bytes) is moved over the message m and the windows' contents $m_{p,w}$ are used as input for the hash function H_R to compute a pseudorandom *fingerprint* at every byte position p (the authors of [14] and [18] use a rolling hash function to compute those fingerprints). The positions whose fingerprints are below a specific threshold T are used as chunk boundaries. We make three changes to the original procedure: First, we calculate the fingerprints using $\text{H}_R(k, i, x)$ with $k := K_O$, so that chunking depends on K_O. Secondly, we prevent repeating window contents from resulting in the same fingerprint by including an appearance counter i in the fingerprint calculation. Thirdly, we introduce a minimum chunk size of l bytes. More formally, we create a chunk boundary at every byte position p that meets the following condition:

$$p - l \geq \max_{p',p'<p}\{p' = 0 \vee p' \text{ is a chunk boundary}\} \\ \bigwedge \text{H}_R\left(K_O,\ \sum_{m_{p',w}=m_{p,w},p'<p} 1,\ m_{p,w}\right) < T \qquad (1)$$

By choosing $T = \frac{2^n}{S-l+1}$ with a to-be-chosen parameter value $S \geq l$, Condition 1—applied to random data—is expected to hold for every S-th byte, resulting in chunks with an average length of S bytes. This is indeed also true for non-random data, as will be discussed in Appendix A.

5.1.2 Chunk encryption

Let $m_1, \ldots, m_q$ be the chunks resulting from the chunking procedure described above, numbered in the order of their appearance. For each chunk m_j, we compute a chunk-specific encryption key K_{m_j} which we use to encrypt the chunk's contents. Instead of using a simple hash value of the chunk contents—as done in [18]—we compute this key as follows:

$$K_{m_j} = \text{H}_S\left(K_O,\ \sum_{m_{j'}=m_j,j'<j} 1,\ m_j\right) \qquad (2)$$

Again, the usage of K_O ensures that different keys are computed for equal chunks in different messages, unless the same value K_O was explicitly used for different encryptions—e.g. for different revisions of the same file. To avoid equal encryption of equal chunks within the same message, an appearance counter is used. This counter is 0 for the first and $i-1$ for the i-th occurrence of the same chunk, thus multiple equal chunks get unique keys.

The key K_{m_j} is then used to encrypt the chunk contents using the pseudorandom permutation F. For technical reasons, we prefix the chunk content with its length before encrypting it[4]. K_{m_j} is then encrypted (using the key K_R) and stored within the encrypted chunk representation. To summarize, every chunk m_j is encrypted as follows (with $\|$ being the concatenation operator):

$$c_j = \left(F_{K_R}(K_{m_j}), F_{K_{m_j}}(\text{length}(m_j) \,\|\, m_j)\right) \qquad (3)$$

5.1.3 Finalization

When all chunks have been encrypted, their ciphertexts $c_1, \ldots, c_q$ are concatenated to build the ciphertext c of the whole message:

$$\widehat{\text{Enc}}_{(K_R,K_O)}(m) = \left(\|_{j=1}^{q} c_j\right) \qquad (4)$$

The decryption function $\widehat{\text{Dec}}$ is straightforward: On input a key $K = (K_R, K_O)$ and a ciphertext c, it sequentially decrypts the chunks $c_1, \ldots, c_q$: For every chunk $c_j = (c_j^k, c_j^c)$, it first determines the chunk key $K_{m_j} := F_{K_R}^{-1}(c_j^k)$ and then decrypts the chunk content and length by invoking $l_j \| m_j := F_{K_{m_j}}^{-1}(c_j^c)$. Finally, $\|_{j=1}^{q} m_j$ is output. Note that decryption depends only on K_R, not on K_O.

5.2 Advantages

If every encryption application uses a random K_O, our scheme achieves CCA-secure encryptions, as we prove in Sec. 6.1. However, it can achieve storage efficiency within the SVN scenario when used as follows:

- Assign random keys K_R, K_O, K_I to every file; change them only on access rights changes.

As long as appearance counters are not affected, using the same K_O for multiple file revisions results in a deterministic, context-sensitive chunking, which achieves identical chunks within unchanged parts between those revisions—even if the positions of those unchanged parts change between revisions (i.e. sth. is inserted or removed). Due to the adaption of convergent encryption, these chunks result in the same ciphertext everytime they are encrypted (both chunk key generation and encryption are deterministic). This way, ciphertexts of similar file revisions overlap, so SVN's storage backend can perform data deduplication while neither requiring knowledge about the file's actual contents nor about the specific encryption scheme. As SVN uses differences during update/commit operations (cf. Sec. 2), this also reduces the network traffic between clients and the repository.

If appearance counters are indeed affected by changes made to a file, different file revisions *might* get slightly different chunkings, thus preventing deduplication of some chunks. Our evaluation results (see Appendix A) suggest that the storage overhead caused by this limitation is negligible.

Note that this only allows data deduplication between different revisions of a specific file as long as its access rights are unchanged. Data deduplication across access rights changes can also be achieved by only changing K_R and K_I on access rights changes (avoiding influences on chunking and encryption of chunk contents). However, this scenario would leak some additional information to attackers with access rights to some other revision of a specific file (see Sec. 6.2.2).

[4]When instantiated with a block cipher, this prevents the need for explicit chunk separators in the ciphertext, since decrypting the first few bytes of a chunk allows computing the starting position of the subsequent chunk.

5.3 Implementation Details

While our encryption scheme is independent of its underlying cryptographic functions, its application requires an appropriate choice of these functions. We use SHA-256-HMAC [8] as MAC Π_M and instantiate the random oracle H with SHA-256; queries prefixed by a key are performed using SHA-256-HMAC (with the key used as key instead of a prefix). The pseudorandom permutation F is instantiated with AES-256-CBC [5], with the IVs being chosen as follows to achieve deterministic encryption:

For encryption of the chunk content, we set $IV = 0$. While this is considered to be insecure in general use cases, our content-based key generation guarantees that we never encrypt different data using the same key. This makes the zero vector unique in the scope of the used encryption key.

For encryption of the chunk key, the situation is not that simple: Since the same key K_R is used to encrypt different chunk keys, we cannot use a constant IV here. We solve this by using the first block of the ciphertext of a chunk as IV for the encryption of its corresponding chunk key[5]. Since no two chunks have the same key, these ciphertext blocks differ for different chunks, resulting in a unique IV. Effectively, this makes the first block of the chunk key encryption and the second block of the chunk content encryption depend on the same IV, but since different keys are used for these encryptions, the uniqueness property is retained.

6. SECURITY ANALYSIS

Our encryption scheme's security guarantees depend on its usage: We achieve IND-CCA-secure encryption assuming the existence of a PRP and a random oracle when random values K_O are used for each encryption application[6]. Otherwise, some controlled amount of information is leaked to allow data deduplication. We analyze both variants' security guarantees—regarding attackers that neither know K_R, K_O nor K_I—separately in this section. The security analysis follows the definitions and proofs given in [13].

6.1 IND-CCA with random K_O

To formalize the usage scenario where a random value K_O is used for each encryption application, we consider a slightly modified version Π^* of our encryption and authentication scheme Π, which replaces the encryption part $\widehat{\Pi}$ of the scheme with $\widetilde{\Pi} = (\widetilde{\text{GEN}}, \widetilde{\text{ENC}}, \widetilde{\text{DEC}})$:

- $\widetilde{\text{GEN}}(n)$ invokes $(K_R, K_O) \leftarrow \widehat{\text{GEN}}(n)$ and returns K_R.
- $\widetilde{\text{ENC}}_{K_R}(m)$ invokes $(\_, K_O) \leftarrow \widehat{\text{GEN}}(n)$ to generate a random K_O and returns $\widehat{\text{ENC}}_{(K_R,K_O)}(m)$. (The $\_$ denotes that the first component is ignored.)
- $\widetilde{\text{DEC}}_{K_R}(c)$ simply invokes $m \leftarrow \widehat{\text{DEC}}_{(K_R,\perp)}(c)$ and returns m (note that K_O is not used in decryption).

The only difference between Π and Π^* is that the latter does not include a static value K_O in the key material, but chooses a random value K_O during each encryption.

We want to show that $\widetilde{\Pi}$ and Π^* are CPA-secure and CCA-secure, respectively. For this, we define some variants $\widetilde{\Pi}_i$ of the encryption scheme $\widetilde{\Pi}$. In addition to F, H_R, H_S, we let $\widetilde{\Pi}_i$ have access to 2^n different truly random permutations $f_0(\cdot), \ldots, f_{2^n-1}(\cdot)$ (one truly random permutation corresponds to each key of the PRP F), and define it as follows:

- $\widetilde{\text{GEN}}_i = \widetilde{\text{GEN}}$, $\widetilde{\text{DEC}}_i = \widetilde{\text{DEC}}$
- $\widetilde{\text{ENC}}_i$, on input key K_R and message m, acts like $\widetilde{\text{ENC}}$, but changes the—overall[7]—first i queries to F so that the invocation $F_K(x)$ is replaced by a query $f_K(x)$.

It is easy to see that $\widetilde{\Pi}_0 = \widetilde{\Pi}$. Note that for $i > 0$, $\widetilde{\Pi}_i$ does not represent a correct encryption scheme, as $\widetilde{\text{ENC}}_i$ can produce ciphertexts that $\widetilde{\text{DEC}}_i$ cannot decrypt. We can ignore this since the $\widetilde{\Pi}_i$'s are only auxiliary constructions and our proof does not depend on the existence of $\widetilde{\text{DEC}}_i$.

Theorem. *If H_R, H_S are random oracles, $\widetilde{\Pi}_\infty$ achieves* IND-CPA-secure *encryption.*

Proof. Our first goal is to show that $\widetilde{\Pi}$ achieves IND-CPA-secure encryption if *all* PRP invocations are replaced by invocations of truly random functions. We denote this variant by $\widetilde{\Pi}_\infty$ to indicate that $i \to \infty$.

By [13, Definition 3.21], "*a private-key encryption scheme $\widetilde{\Pi}_i$ is IND-CPA-secure if for all probabilistic polynomial-time adversaries $\mathcal{A}$ there exists a negligible function* negl *such that* $\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_i}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n)$*, where the probability is taken over the random coins used by $\mathcal{A}$, as well as the random coins used in the experiment.*" The experiment $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_i}(n)$ is defined as follows in [13, Section 3.5] (for convenience, we adapt the presentation to our notations):

1. A key K is generated by running $\widetilde{\text{GEN}}_i(1^n)$.
2. The adversary $\mathcal{A}$ is given input 1^n and oracle access to $\widetilde{\text{ENC}}_{i_K}(\cdot)$, and outputs a pair of messages m_0, m_1 of the same length.
3. A random bit $b \leftarrow \{0,1\}$ is chosen, and then a ciphertext $c \leftarrow \widetilde{\text{ENC}}_{i_K}(m_b)$ is computed and given to $\mathcal{A}$. We call c the challenge ciphertext.
4. The adversary $\mathcal{A}$ continues to have oracle access to $\widetilde{\text{ENC}}_{i_K}(\cdot)$, and outputs a bit b'.
5. The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise. (In case $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_i}(n) = 1$, we say that $\mathcal{A}$ succeeded.)

Now let $\mathcal{A}$ be any adversary solving this experiment in polynomial time $q(n)$ (with $q(n)$ being the number of steps $\mathcal{A}$ may perform—including queries to $\widetilde{\text{ENC}}_{i_K}$, H_R, H_S and $f_0, \ldots, f_{2^n-1}$). The intuition of our proof is as follows: $\mathcal{A}$ can only learn sth. about the challenge, if he sees any of the (random) values output by H_R, H_S, f_k during encryption of m_b at any other point of time. We show that this happens with negligible probability.

Let HQ denote the event that any query to H_R or H_S during encryption of m_b in step 2 is also made at any other point of time during the experiment. Similarly, let FQ denote the event that a specific query $f_k(x)$ during encryption of m_b occurs at any other point of time. If all of those queries are unique (i.e. $\overline{\mathsf{HQ} \vee \mathsf{FQ}}$), $\mathcal{A}$ has never seen any of the values output by H_R (cf. Eq. 1), H_S (cf. Eq. 2) or any f (cf. Eq. 3) that are used to produce the ciphertext of m_b. As H_R, H_S are modelled using a random oracle and as f_k produces truly random outputs, those values are independent from m_0, m_1 in $\mathcal{A}$'s view. Thus, $\mathcal{A}$'s chance to succeed is $\frac{1}{2}$:

$$\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_\infty}(n) = 1 \mid \overline{\mathsf{HQ} \vee \mathsf{FQ}}] = \frac{1}{2} \tag{5}$$

Repetitions of queries to H_R, H_S within a single encryption are excluded due to the appearance counters. As each

[5]This also saves us the need for storing the IV separately.

[6]This always applies to files with only a single revision.

[7]This means the first i queries during $\widetilde{\Pi}_i$'s whole lifetime.

query to H_R and H_S includes the key K_O, which is chosen uniformly at random during encryption, a repetition of some query to H_R or H_S requires that the same value K_O was either chosen by chance during another encryption application or that it was guessed by $\mathcal{A}$ during his runtime. The chance for this to happen within runtime $q(n)$ is:

$$\Pr[\mathsf{HQ}] \leq \frac{q(n)}{2^n} \tag{6}$$

We now analyze the probability of FQ under $\overline{\mathsf{HQ}}$ (i.e. the case where all H_R, H_S queries are unique). A repetition of a query to some f_k during encryption of m_b can occur either during other encryptions or due to direct queries by $\mathcal{A}$. Both cases are analyzed separately in the following paragraphs:

$\widetilde{\text{ENC}}_{i_K}$ queries the truly random permutations by $f_K(j)$ and $f_j(\cdot)$ for every output j produced by H_S (i.e. for each chunk key, cf. Eq. 3). Thus, if all outputs j of H_S are unique, all queries $f_K(j), f_j(\cdot)$ are unique. As we assume $\overline{\mathsf{HQ}}$, no repetition of a query to H_S occurred, so a repetition of a query to any f_k during encryption requires that at least two different queries to H_S produced equal outputs during the experiment. As $\mathcal{A}$ has runtime $q(n)$, he can only query the encryption of messages of length at most $q(n)$. During encryption of a message of length $q(n)$, at most $q(n)$ queries are made to H_S, resulting in a maximum number of $q(n)^2$ queries to H_S during the whole experiment. As H_S generates outputs of length n bits, the collision probability of H_S during the experiment is bounded by $\frac{q(n)^2 2}{2 \cdot 2^n}$.

A repetition of a query to f_k might also occur due to an explicit query by $\mathcal{A}$. However, if no repetition of a query to H_S occurred, this requires that $\mathcal{A}$ correctly guesses at least one chunk key j for which a query $f_K(j)$ or $f_j(\cdot)$ is made by $\widetilde{\text{ENC}}_{i_K}$ during encryption of m_b. As $\mathcal{A}$ has runtime $q(n)$ and as there are at most $q(n)$ chunk keys each having a length of n bits, the chance for this to happen is bounded by $\frac{q(n)^2}{2^n}$.

We conclude:

$$\Pr[\mathsf{FQ} \mid \overline{\mathsf{HQ}}] \leq \frac{q(n)^2 2}{2 \cdot 2^n} + \frac{q(n)^2}{2^n} = \frac{q(n)^4 + 2 \cdot q(n)^2}{2^{n+1}} \tag{7}$$

Combining Equations 5, 6 and 7, we get:

$$\begin{aligned}
& \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_\infty}(n) = 1] \\
= \; & \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_\infty}(n) = 1 \mid \overline{\mathsf{HQ} \vee \mathsf{FQ}}] \cdot \Pr[\overline{\mathsf{HQ} \vee \mathsf{FQ}}] \\
& + \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_\infty}(n) = 1 \mid \mathsf{HQ} \vee \mathsf{FQ}] \cdot \Pr[\mathsf{HQ} \vee \mathsf{FQ}] \\
\leq \; & \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_\infty}(n) = 1 \mid \overline{\mathsf{HQ} \vee \mathsf{FQ}}] + \Pr[\mathsf{HQ} \vee \mathsf{FQ}] \\
\leq \; & \frac{1}{2} + \Pr[\mathsf{HQ}] + \Pr[\mathsf{FQ}] \\
= \; & \frac{1}{2} + \Pr[\mathsf{HQ}] + \Pr[\mathsf{FQ} \mid \mathsf{HQ}] \cdot \Pr[\mathsf{HQ}] + \Pr[\mathsf{FQ} \mid \overline{\mathsf{HQ}}] \cdot \Pr[\overline{\mathsf{HQ}}] \\
\leq \; & \frac{1}{2} + 2\Pr[\mathsf{HQ}] + \Pr[\mathsf{FQ} \mid \overline{\mathsf{HQ}}] \;\leq\; \frac{1}{2} + \frac{2q(n)}{2^n} + \frac{q(n)^4 + 2q(n)^2}{2^{n+1}} \\
= \; & \frac{1}{2} + \frac{q(n)^4 + 2 \cdot q(n)^2 + 4 \cdot q(n)}{2^{n+1}} \;\leq\; \frac{1}{2} + \mathsf{negl}(n)
\end{aligned} \tag{8}$$

This proves that $\widetilde{\Pi}_\infty$ is IND-CPA-secure. □

Theorem. *If F is a pseudorandom permutation and H_R, H_S are random oracles, $\widetilde{\Pi}$ achieves* CPA-secure *encryption.*

Proof. We already know that $\widetilde{\Pi}$ achieves CPA-secure encryption if only truly random permutations are used. We will show that $\mathcal{A}$'s advantage is negligible if only single invocations of some truly random permutation are replaced by a PRP. As there is only a polynomial amount of invocations, we conclude that $\widetilde{\Pi}$ is also CPA-secure using only a PRP.

Regard any polynomially bounded adversary $\mathcal{A}$ with runtime $q(n)$. Our goal is to show that $\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n)$. We know that $\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_\infty}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n)$. Due to his runtime $q(n)$, $\mathcal{A}$ can make at most $q(n)$ queries to $\widetilde{\text{ENC}}_{i_K}$, with each queried message having a maximum length of $q(n)$. As the encryption operation performs at most one query to F_k and f_k, respectively, for every byte of its input, a maximum of $2q(n)^2$ queries to F_k/f_k may occur within the whole experiment. Within this runtime, $\widetilde{\Pi}_\infty$ and $\widetilde{\Pi}_{2q(n)^2}$ behave equally, so we have:

$$\begin{aligned}
\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_{2q(n)^2}}(n) = 1] &= \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_\infty}(n) = 1] \\
&\leq \frac{1}{2} + \mathsf{negl}(n)
\end{aligned} \tag{9}$$

Next we show that $\mathcal{A}$ cannot distinguish the experiments $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_i}(n)$ and $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_{i-1}}(n)$ with non-negligible probability. We do this by defining a polynomial-time distinguisher $D^{F'(\cdot)}$ as follows:

1. Run $\mathcal{A}(1^n)$ and simulate the oracle $\widetilde{\text{ENC}}_{i_K}(\cdot)$ by executing the algorithm $\widetilde{\text{ENC}}_{i_K}$ to answer $\mathcal{A}$'s oracle queries—with one modification: When the—overall—i-th query to a (pseudo-)random permutation (F or f) is made, use F' instead of $f_K(x)$.
2. When $\mathcal{A}$ outputs messages m_0, m_1, choose $b \leftarrow \{0,1\}$ randomly and return $\widetilde{\text{ENC}}_{i_K}(m_b)$.
3. Continue answering oracle queries by $\mathcal{A}$ as in step 1.
4. When $\mathcal{A}$ outputs b', output 1 if $b' = b$, otherwise 0.

If D is instantiated with the PRP F', the only difference between D and experiment $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_{i-1}}(n)$ is that $F'(x)$ is queried instead of $F_j(x)$ at any point of time. As there might be other queries to F_j during the experiment, two situations might lead to inconsistent behavior: $F_j(x)$ might be queried at any other point of time during the experiment and produce different output than $F'(x)$, or $F'(x)$ might produce an output that is also output by $F_j(y), y \neq x$ at any other point of time during the experiment. As the experiment allows at most $2q(n)^2$ invocations of F_j, the probability of the latter is upper-bounded by $\frac{2q(n)^2}{2^n}$. Further, as all invocations $F_j(x)$ done by $\widetilde{\text{ENC}}_{i_K}$ include an n-bit number chosen uniformly at random (cf. Equation 3), a repetition of the invocation $F_j(x)$ requires that this n-bit value has been chosen again by chance or has been guessed by $\mathcal{A}$. The probability for this to happen is upper-bounded by $\frac{2q(n)^2}{2^n}$, too.

If neither of these situations occur, we can think of $D^{F'(\cdot)}$ as using a slightly modified PRP $\widetilde{F}$ (for the whole experiment) that is defined exactly like F, but switches two values so that its outputs are consistent to F': $\widetilde{F}_j(x) = F'(x)$ and $\widetilde{F}_j(F_j^{-1}(F'(x))) = F_j(x)$. As $\widetilde{F}$ is a PRP, the view of $\mathcal{A}$ when run by D is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_{i-1}}(n)$. As the probability for this situation is at least $1 - \frac{4q(n)^2}{2^n}$ and D outputs 1 whenever $\mathcal{A}$ succeeds in the experiment, we know that:

$$\Pr[D^{F'(\cdot)}(1^n) = 1] \geq \left(1 - \frac{4q(n)^2}{2^n}\right) \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_{i-1}}(n) = 1] \tag{10}$$

If D is instantiated with a truly random permutation f', the only difference between D and experiment $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_i}(n)$

is that $f'(x)$ is queried instead of $f_j(x)$ at any point of time. We can use the same arguments as before to show that the view of $\mathcal{A}$ when run by D is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_i}(n)$ with probability at least $1 - \frac{4q(n)^2}{2^n}$. Assuming that $\mathcal{A}$ succeeds in any other case, we can give an upper bound:

$$\Pr[D^{f'(\cdot)}(1^n) = 1] \leq \left(1 - \frac{4q(n)^2}{2^n}\right) \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_i}(n) = 1] + \frac{4q(n)^2}{2^n} \quad (11)$$

As we know (by assumption) that F' is a PRP, we get:

$$\begin{aligned}\mathsf{negl}'(n) &\geq \Pr[D^{F'(\cdot)}(1^n) = 1] - \Pr[D^{f'(\cdot)}(1^n) = 1] \\ &\geq \left(1 - \frac{4q(n)^2}{2^n}\right) \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_{i-1}}(n) = 1] \\ &\quad - \left(1 - \frac{4q(n)^2}{2^n}\right) \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_i}(n) = 1] - \frac{4q(n)^2}{2^n}\end{aligned}$$

We can solve this to get:

$$\begin{aligned}\mathsf{negl}''(n) &\geq \left(\mathsf{negl}'(n) + \frac{4q(n)^2}{2^n}\right) \cdot \frac{1}{1 - \frac{4q(n)^2}{2^n}} \\ &\geq \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_{i-1}}(n) = 1] - \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_i}(n) = 1] \quad (12)\end{aligned}$$

Combining Equations 9 and 12, we conclude:

$$\begin{aligned}&\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1] = \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_0}(n) = 1] \\ \leq\ &\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_{2q(n)^2}}(n) = 1] \\ &+ \left|\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_0}(n) = 1] - \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_{2q(n)^2}}(n) = 1]\right| \\ \leq\ &\frac{1}{2} + \mathsf{negl}(n) \\ &+ \sum_{i=1}^{2q(n)^2} \left|\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_{i-1}}(n) = 1] - \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}_i}(n) = 1]\right| \\ \leq\ &\frac{1}{2} + \mathsf{negl}(n) + 2q(n)^2 \cdot \mathsf{negl}''(n) \leq \frac{1}{2} + \mathsf{negl}'''(n) \quad (13)\end{aligned}$$

This proves that $\widetilde{\Pi}$ is CPA-secure. □

Theorem. *If $\widetilde{\Pi}$ achieves IND-CPA-secure encryption, Π^* achieves* IND-CCA-secure *encryption.*

Proof. As Π^* is an instantiation of [13, Construction 4.19] (see Sec. 5.1), we can apply [13, Theorem 4.20], whose proof can be found in [13, Chapter 4.8]: "*If Π_E is a CPA-secure private-key encryption scheme and Π_M is a secure message authentication code with unique tags, then Construction 4.19 is a CCA-secure private-key encryption scheme.*" Since $\widetilde{\Pi}$ is CPA-secure and Π_M is a secure message authentication code with unique tags, Π^* is CCA-secure. □

6.2 Security Guarantees with fixed K_O

If K_O is re-used for multiple encryptions (in our system, this can be the case for different revisions of the same file), we intentionally leak some information to allow data deduplication. We allow a possible attacker to see relations between different messages that are encrypted with the same K_O, so he might recognize the positions and the size of plaintext fragments that are identical between different messages. This shall even be true if equal plaintext fragments appear at different positions within different messages.

This is a rather specific use case which prevents the application of classic security properties like ciphertext indistinguishability. However, we want to show that re-using K_O retains some basic notion of security. While we leak information about *identical* contents in different messages, we can prove that we do not leak any information about *different* contents that are encrypted using the same value K_O.

6.2.1 Security Guarantees for Different Plaintexts

To formalize this, we introduce a restricted variant of the $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widehat{\Pi}}(n)$ experiment seen in the previous section: the *chosen different plaintext attack (CDPA)* experiment. $\mathsf{PrivK}^{\mathsf{cdpa}_d}_{\mathcal{A},\widehat{\Pi}}(n)$ is defined exactly like $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widehat{\Pi}}(n)$, with the difference that $\mathcal{A}$ is only allowed to ask for encryptions of messages which do not have an overlapping fragment with more than d bytes size. Thus, all substrings occuring in more than one message (including both queries $\mathcal{A}$ makes to $\widehat{\mathrm{ENC}}_K$ as well as the messages m_0, m_1) must have a maximum size of d bytes. If $\mathcal{A}$ does not adhere to these constraints, the output of the experiment is defined to be 0.

Analogous to the previous definition, we say an encryption scheme $\widehat{\Pi}$ is *CDPA$_d$-secure* if for all probabilistic polynomial-time adversaries $\mathcal{A}$ there exists a negligible function negl such that $\Pr[\mathsf{PrivK}^{\mathsf{cdpa}_d}_{\mathcal{A},\widehat{\Pi}}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n)$. Again, the probability is taken over the random coins used by $\mathcal{A}$ and the random coins used in the experiment.

We emphasize that this is a *significantly weaker* security model than CPA security, since we cannot hope to show strong security properties when intentionally leaking information to an attacker. The security proof shall merely provide an intuition that our scheme does not leak information in situations where we do not explicitly require it to do so.

Theorem. *If $\widetilde{\Pi}$ achieves CPA-secure encryption, $\widehat{\Pi}$ achieves* CDPA$_d$-secure *encryption for $d = \min\{w, l\} - 1$.*

Proof. To show this, we need to define another variant of our encryption scheme $\widehat{\Pi}$ first. Let $\widehat{\Pi}_i = (\widehat{\mathrm{GEN}}_i, \widehat{\mathrm{ENC}}_i, \widehat{\mathrm{DEC}}_i)$ be defined as follows:

- $\widehat{\mathrm{GEN}}_i = \widehat{\mathrm{GEN}}$, $\widehat{\mathrm{DEC}}_i = \widehat{\mathrm{DEC}}$
- $\widehat{\mathrm{ENC}}_i$, on input key (K_R, K_O), invokes $(\_, K_{O'}) \leftarrow \widehat{\mathrm{GEN}}_i(n)$ to obtain a random $K_{O'}$. Then it acts like $\widehat{\mathrm{ENC}}$, but uses $K_{O'}$ instead of K_O for the—overall—first i queries to $H_X, X \in \{R, S\}$ that it makes during its lifetime.

Now regard the sequence of encryption schemes $\widehat{\Pi}_0, \widehat{\Pi}_1, \ldots$ Surely, $\widehat{\Pi}_0$ behaves exactly like $\widehat{\Pi}$, so we have:

$$\Pr[\mathsf{PrivK}^{\mathsf{cdpa}_d}_{\mathcal{A},\widehat{\Pi}_0}(n) = 1] = \Pr[\mathsf{PrivK}^{\mathsf{cdpa}_d}_{\mathcal{A},\widehat{\Pi}}(n) = 1] \quad (14)$$

We can further see that—up to the i-th oracle query—$\widehat{\mathrm{ENC}}_i$ behaves exactly like $\widehat{\mathrm{ENC}}$, since it uses a fresh, randomly generated key $K_{O'}$ for each encryption application instead of the supplied value K_O.

Now let $q(n)$ be the maximum runtime of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{cdpa}_d}_{\mathcal{A},\widehat{\Pi}}(n)$. In each step, $\mathcal{A}$ can make at most one query to the encryption oracle, so $\widehat{\mathrm{ENC}}$ can be executed at most $q(n)$ times during that experiment. During encryption of a message m, $\widehat{\mathrm{ENC}}$ makes at most $|m|$ queries to H_R and at most $|m|$ queries to H_S, respectively. As—due to its runtime—$\mathcal{A}$ cannot generate plaintexts of length greater than $q(n)$, each encryption query results in at most $2q(n)$ queries to $H_X, X \in \{R, S\}$, resulting in a maximum total

number of $2q(n)^2$ queries to H_X during the whole experiment. As $\widehat{\text{ENC}}_i$ and $\widetilde{\text{ENC}}$ are identical up to the i-th query to H_X, this implies:

$$\Pr[\mathsf{PrivK}^{\mathsf{cdpa}_d}_{\mathcal{A},\widehat{\Pi}_{2q(n)^2}}(n)=1] \;=\; \Pr[\mathsf{PrivK}^{\mathsf{cdpa}_d}_{\mathcal{A},\widetilde{\Pi}}(n)=1] \qquad (15)$$

We show that an adversary cannot distinguish the experiments $\mathsf{PrivK}^{\mathsf{cdpa}_d}_{\mathcal{A},\widehat{\Pi}_i}(n)$ and $\mathsf{PrivK}^{\mathsf{cdpa}_d}_{\mathcal{A},\widehat{\Pi}_{i-1}}(n)$ with non-negligible probability: The only difference between both experiments is that $H_X(K_{O'},x,y)$ is queried instead of $H_X(K_O,x,y)$ at any point of time within the former experiment.

Per definition, $\mathcal{A}$ can only succeed if all queries made to the encryption oracle during the experiment are different in the sense that no $\min\{w,l\}$-byte window content occurs in more than one query. Thus, different success probabilities for those experiments imply that $\mathcal{A}$ adheres to those constraints. As w-byte window contents are used as input to H_R by the encryption scheme (see Equation 1) and repetitions of the same w-byte window within a single encryption application result in different inputs to H_R due to the appearance counter, this implies that all inputs to H_R during the whole experiment are different. Similarly, as the encryption scheme does not generate chunks smaller than l bytes, all inputs to H_S (see Equation 2) are unique, too.

If $\mathcal{A}$ did not query any of the values $H_X(K_{O'},x,y)$ or $H_X(K_O,x,y)$ directly to the oracle, both values are—due to the random oracle—independent from both all inputs and all other outputs of the random oracle, so $\mathcal{A}$ cannot distinguish between those values. Different success probabilities in both experiments thus require that $\mathcal{A}$ did query any of those values within the experiment. As this requires a correct guess of either K_O or $K_{O'}$ within $\mathcal{A}$'s runtime, the probability for this to happen is at most $\frac{2q(n)}{2^n}$:

$$\left|\Pr[\mathsf{PrivK}^{\mathsf{cdpa}_d}_{\mathcal{A},\widehat{\Pi}_i}(n)=1]-\Pr[\mathsf{PrivK}^{\mathsf{cdpa}_d}_{\mathcal{A},\widehat{\Pi}_{i-1}}(n)=1]\right| \le \frac{2q(n)}{2^n} \qquad (16)$$

Combining Equations 14, 15 and 16, we get:

$$\left|\Pr[\mathsf{PrivK}^{\mathsf{cdpa}_d}_{\mathcal{A},\widetilde{\Pi}}(n)=1]-\Pr[\mathsf{PrivK}^{\mathsf{cdpa}_d}_{\mathcal{A},\widehat{\Pi}}(n)=1]\right| \le\; 2q(n)^2\cdot\frac{2q(n)}{2^n} \;=\; \frac{q(n)^3}{2^{n-2}} \qquad (17)$$

The only difference between $\mathsf{PrivK}^{\mathsf{cdpa}_d}_{\mathcal{A},\widetilde{\Pi}}(n)$ and $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n)$ is that $\mathcal{A}$ is less restricted in the latter. We conclude:

$$\begin{aligned}\Pr[\mathsf{PrivK}^{\mathsf{cdpa}_d}_{\mathcal{A},\widehat{\Pi}}(n)=1] \;&\le\; \Pr[\mathsf{PrivK}^{\mathsf{cdpa}_d}_{\mathcal{A},\widetilde{\Pi}}(n)=1]+\frac{q(n)^3}{2^{n-2}}\\ &\le\; \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n)=1]+\frac{q(n)^3}{2^{n-2}} \;\le\; \frac{1}{2}+\mathsf{negl}(n)+\frac{q(n)^3}{2^{n-2}}\\ &\le\; \frac{1}{2}+\mathsf{negl}'(n) \qquad (18)\end{aligned}$$

with the last inequality being true because $q(n)$ is polynomially bounded. This shows that $\widehat{\Pi}$ is CDPA_d-secure. □

Theorem. *If $\widehat{\Pi}$ achieves CDPA_d-secure encryption, Π also achieves* CDPA_d-secure *encryption.*

Proof. The only difference between $\widehat{\Pi}$, which we have proven to be CDPA_d-secure, and Π is that the latter appends a MAC tag generated by Π_M to each ciphertext. As the calculation of this tag does only depend on the ciphertext and an independent key, its presence surely cannot reduce the security properties provided by $\widehat{\Pi}$. We conclude that Π is CDPA_d-secure, too. □

Combining both theorems, we have proven that Π achieves $\text{CDPA}_{\min\{w,l\}-1}$-secure encryption. Intuitively, this means, that an arbitrary amount of plaintexts may be encrypted using the same (K_R,K_O,K_I) key triple without leaking any information to an attacker, as long as none of those plaintexts share a substring of length $\min\{w,l\}$ bytes.

6.2.2 Security Guarantees for Similar Plaintexts

While the model presented in the previous section provides strong security guarantees under certain conditions, we have to intentionally leak some information to achieve storage efficiency. In other words, the conditions required for the proof cannot always be met. We therefore describe what information an attacker can gain in the general case.

It is easy to see that identical plaintexts are always encrypted to the same ciphertext (if K_R, K_O and K_I are unchanged), so in this case, an attacker can see nothing more than *how often* a specific plaintext was encrypted.

The encryption of different, but similar plaintexts (that share a substring of length at least $\min\{w,l\}$ bytes) using the same key is the most common use case. We analyze what information is leaked in this case to provide an intuition that our design decisions are reasonable. The analysis is performed in two steps: First we focus on security implications of our chunk encryption scheme, later we analyze the chunking procedure.

Security Implications of Chunk Encryption

Imagine all chunk boundaries have been determined completely randomly. Let $m^1,\ldots,m^M$ be the list of all messages that have been encrypted with $\widehat{\Pi}$ using the same key and let $c^1,\ldots,c^M$ be their respective ciphertexts. Let further m^j_i denote the i-th chunk of message m^j and let c^j_i denote its encrypted representation. Clearly, our deterministic encryption mechanism leaks information about equality of chunks[8]. But what information is leaked beyond equality?

To analyze this, regard the sequence of all *different* chunks $m'_1,\ldots,m'_r$ that have ever been encrypted using the same key and let C'_i denote the maximum number of occurrences of m'_i within any *single* message. From those chunks, we can build a plaintext $m=\|_{i=1}^{r}\left(\|_{j=1}^{C'_i} m'_i\right)$. If we apply a modified version of $\widetilde{\Pi}$, that achieves exactly that chunking, to m, we get a ciphertext c that contains all encrypted chunks that are contained in $c^1,\ldots,c^M$. Thus, $c^1,\ldots,c^M$ can be reconstructed from c just using knowledge about *which substrings* have to be concatenated in *which order*, so all non-positional information leaked by $c^1,\ldots,c^M$ is leaked by c, too.

In this setup, however, we have only a single application of our encryption scheme that uses a random value K_O, which is CPA-secure as shown in Section 6.1. We conclude that chunk encryptions do not leak information beyond *positions* and *sizes* of contents.

If an attacker knows K_O (e.g. because he has read access to another revision that uses the same K_O), he can encrypt plaintexts (and also determine chunk boundaries) himself, allowing a verification of guessed plaintexts. Therefore, the default behaviour of our implementation is to change K_O on access rights changes.

[8]This applies across revisions of the same file with equal K_O, not within one revision due to the appearance counters.

Security Implications of Chunking

We have already seen that an encrypted chunk for itself does not leak any information about its underlying plaintext. However, due to the context-sensitive chunking procedure, we can think of each chunk as being annotated with some information about its plaintext visible to an adversary. To see which information is leaked, regard an arbitrary but fixed chunk c_i. From the chunking mechanism, an attacker gets to know the following facts:

- If c_i occurs within any message at any position other than its beginning, the attacker knows that the first w bytes of the chunk fulfill Condition 1.
- The attacker knows that no w-bytes substring within the last $|c_i|-l$ bytes of c_i fulfill Condition 1 within any yet encrypted message that contains c_i.

We emphasize that an adversary cannot evaluate Condition 1 for any plaintext without knowledge about K_O, so this information does not obviously allow to draw conclusions about the plaintext. However, when two revisions of a file are encrypted using the same K_O, limited structural information about that file might be leaked: Chunking may reveal the positions of changes more precisely than the encryption procedure itself, if a change affects a chunk boundary. In addition, changed chunk boundaries might affect chunk appearance counters and thus the encryption of later equal chunks, so that those chunk ciphertexts might appear to *move* between two revisions. However, both potential issues are in line with our security claims, which allow revealing the positions and extent of changes if K_O is unchanged.

7. SYSTEM IMPLEMENTATION

We have implemented the full concept, as described in the previous sections, into the SVN library source code[9], extended the SVN command line application, and evaluated the performance of our implementation.

We wanted to retain full compatibility to other SVN versions to enable a quick deployment of our extension. We have therefore realized nearly all parts of our solution on the client side—namely the *working copy library*—of the SVN architecture without introducing any new data structures that would have to be handled by the repository. For this, we store all data in so-called *properties*—a versioned file-related meta-data mechanism provided by the SVN architecture. All actions provided by our solution (e.g. encryption/decryption of files, permission administration) are designed to only affect a file's representation or its properties within the user's working copy, while the synchronization between a working copy and the repository (e.g. commit/update operations) stays unchanged.

Key management is implemented as follows: All keys are generated randomly when a confidential file is created or access rights are changed. To allow data deduplication, we do not automatically change a file's K_O when new revisions are generated. A simple command line option allows to achieve stronger security guarantees by explicitly changing K_O.

Regarding compatibility, we support arbitrary combinations between old/new server (repository) versions and old/new client versions. If a client supports our extension, it can securely use our access control solution on some files no matter what other SVN versions are involved at the server and at the client side. Clients not supporting our solution would just be unable to access confidential files—just like new SVN clients that are not granted rights on those files. If the server runs an old version, only server-side checks like write access control would be disabled, so other clients would be able to delete confidential files. However, such files could still be restored due to the version history provided by SVN.

[9]We extended revision 1152561 of the repository's trunk (`https://svn.apache.org/repos/asf/subversion/trunk`)

8. CONCLUSION AND OUTLOOK

We have presented a security solution for the version control system SVN, which enables secure and storage-efficient versioning of documents—even in case of a malicious repository server administrator. The main restrictions of the system are the attacker's capability of seeing the positions of changes in a document, and of verifying whether chunks from a previous version (to which the attacker had been granted access) are still contained in a new version. Both attacks can be prevented by changing the file's *obfuscator*, but this comes at the cost of storage efficiency. Our implementation does not require changes to SVN's storage backend, and is compatible with previous SVN servers (with the exception of write access control) and clients (though old clients cannot access encrypted files). Authentication currently relies on passwords only, as we wanted to avoid the administrative overhead of a PKI. For corporate environments, certificate-based authentication may still be a viable alternative, which we aim at supporting in the future.

9. ACKNOWLEDGMENTS

We thank Gennadij Liske for his valuable comments on the proof (Sec. 6). We further thank Ronald Petrlic, Sebastian Seitz and the anonymous reviewers for their helpful comments. The work is funded by the German Research Foundation (DFG) under GRK 1479.

10. REFERENCES

[1] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Proceedings of CCS '93*, pages 62–73. ACM, 1993.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[3] CollabNet, Inc. Skip-Deltas in Subversion. `http://svn.apache.org/repos/asf/subversion/trunk/notes/skip-deltas`, Nov. 2005.

[4] CollabNet, Inc. The Subversion protocol. `http://svn.apache.org/repos/asf/subversion/trunk/subversion/libsvn_ra_svn/protocol`, Sept. 2011.

[5] J. Daemen and V. Rijmen. AES Proposal: Rijndael. 1999.

[6] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[7] L. Dusseault. HTTP Extensions for Web Distributed Authoring and Versioning. RFC 4918, 2007.

[8] D. E. Eastlake 3rd and T. Hansen. US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). RFC 6234, 2011.

[9] K. Fu, S. Kamara, and T. Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *Proceedings of NDSS*, 2006.

[10] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *Proceedings of NDSS*, 2003.

[11] D. Grolimund, L. Meisser, S. Schmid, and R. Wattenhofer. Cryptree: A folder tree structure for cryptographic file systems. In *25th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 189–198, 2006.

[12] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 29–42, 2003.

[13] J. Katz and Y. Lindell. *Introduction to Modern Cryptography: Principles and Protocols*. Chapman & Hall/CRC, 2008.

[14] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of SOSP '01*, pages 174–187. ACM, 2001.

[15] C. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick. *Version Control with Subversion*. O'Reilly, 2008.

[16] M. O. Rabin. Fingerprinting by Random Polynomials. Technical Report TR-15-81, Department of Computer Science, Harvard University, 1981.

[17] F. Rashid, A. Miri, and I. Woungang. A secure data deduplication framework for cloud environments. In *Tenth Annual International Conference on Privacy, Security and Trust (PST), 2012*, pages 81–87, 2012.

[18] M. W. Storer, K. Greenan, D. D. E. Long, and E. L. Miller. Secure Data Deduplication. In *Proceedings of StorageSS '08*, pages 1–10. ACM, 2008.

[19] The Apache Software Foundation. Apache Subversion. `http://subversion.apache.org/`, Apr. 2012.

[20] Z. Wilcox-O'Hearn and B. Warner. Tahoe: the least-authority filesystem. In *Proceedings of StorageSS '08*, pages 21–26. ACM, 2008.

APPENDIX

A. PERFORMANCE EVALUATION

In this section, we evaluate our solution's performance regarding memory and time efficiency. We first discuss an appropriate choice of the parameter values and then analyze their impact on storage efficiency to prove our savings in comparison to usual encryption schemes. Finally, we briefly evaluate our algorithm's memory and time requirements.

A.1 Choice of Parameters

As mentioned in Section 5.1.1, our encryption scheme depends on some parameter values w (window size), l (minimum chunk size) and S (target chunk size). We set $w = 48$ bytes according to the evaluation results provided by Muthitacharoen et al. [14]. For a random file of size z bytes (and with parameter value $l = 1$ byte), we calculated an average encryption overhead of $\frac{43.5z}{S} + 32$ bytes for storing chunk metadata (length and key), padding and the integrity value. We confirmed this formula to be true for non-random data by encrypting the 50 most popular ebooks (47.9 MiB in total) from Project Gutenberg[10], each as UTF8-encoded text file, with randomly chosen keys with values of S in interval $[32, 4096]$, which resulted in an average deviation of 0.07 percentage points and a maximum deviation of 1.24 percentage points from the expected relative overhead. As our security guarantees depend on $\min\{w, l\}$ (see Sec. 6.2), we set $l = w$, which in addition guarantees a maximum storage overhead of $z + 32$ bytes. With $S \geq 256$, we produce an overhead of $< 20\%$, which we consider an acceptable overhead that we expect to be compensated by the savings due to difference-based efficient storage of multiple file revisions.

[10] `http://www.gutenberg.org/ebooks/search.html/?sort_order=downloads`, visited on 2012-03-13

A.2 Storage Efficiency

While we have shown that an appropriate parameter choice allows storing single file versions with little overhead, our main goal is the efficient storage of whole repositories containing multiple (and similar) revisions of several files. Our system requires storage overhead at several locations. The main overhead—as described before—is caused by the metadata generated by our encryption scheme and depends on the file's size as well as the parameter value S. In addition, since we store all access-control-relevant information (such as cryptographic material) in the confidential file's properties, every confidential file requires some storage for these properties. The storage requirement for this can be quantified with about 1.5 KiB constant overhead per file and 2 KiB constant overhead per authorized user of this file. Note, however, that thanks to differential storage this is only generated once per access right change (at least once per file), not once per revision. Despite that, we also generate small overhead when storing different file revisions: While for unencrypted files, only the contents that actually changed (+ a negligible overhead) have to be stored, our solution requires storage for each change's full surrounding chunk(s).

To evaluate our achievements regarding storage efficiency of similar file revisions, we first studied our algorithm's performance on a kind of best-case scenario, namely a repository that contains the version history of a single file, with only small changes made between each of its revisions. We simulated this situation by setting up an experiment as follows: At first, we committed an empty, confidential file to a fresh SVN repository. Afterwards, we iterated the following sequence: We chose a position within that file uniformly at random and inserted a random number (chosen uniformly at random in interval $[64, 192]$, i.e. 128 bytes on average) of random bytes there. The resulting extended file version was then committed to the repository, so the file and its version history grew with each iteration. Using the generated repository, we compared our encryption scheme to other solutions: For this purpose, we generated a couple of fresh repositories and re-enacted the previously described repository's version history for each of them with individual configurations. In the first configuration, we achieved confidentiality by encrypting each file revision using a traditional encryption scheme (AES-CBC with a fixed key, but randomly chosen IVs); in the remaining ones, we used our access control solution with different parameter values S.

The results of this experiment are shown in Fig. 1. Each line represents the development of the total storage requirement of a specific repository when storing the first i revisions. The black line (+ markers) shows the unencrypted repository, which unsurprisingly has the least storage consumption. The violet line ($\times$ markers) shows the repository whose content is encrypted with a traditional scheme. As expected, its storage consumption rises rapidly with an in-

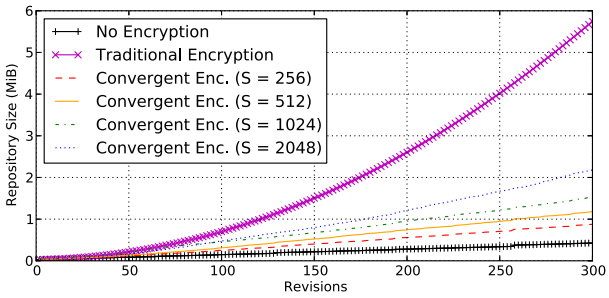

Figure 1: Development of repository size when a versioned confidential file is extended systematically

creasing number of revisions due to the need for storing the whole ciphertext of each file version. While the storage efficiency of our encryption scheme varies for different values of S, its savings are significant for either value. $S = 256$ yields the best performance and requires about twice as much storage as the unencrypted configuration. This is in line with our expectation that small (e.g. about-128-byte) changes result in about S bytes of difference in ciphertext on average.

These results could suggest that a lower S results in lower storage consumption in general, so we repeated the experiment with changes of average length 4 096 bytes instead of 128 bytes between revisions. In this setup, our solution produced less overhead and our savings compared to traditional encryption were more significant. However, savings through small chunk sizes (i.e. small differences between ciphertexts) were outweighed by overhead for storing chunk metadata. $S = 512$ yields the best results in that experiment.

While these results show that our encryption scheme allows for efficient storage in some hypothetic best-case scenario, we surely have to verify if these results are transferable into practice. In fact, we identified two problematic situations for our access control solution. The first is a repository consisting of a huge amount of very small files: Since we have to store metadata for each of these files separately, our system would generate a lot of overhead, which might not be compensated by further savings if the file sizes are not considerably greater than our chunk sizes. The second case is a repository consisting of arbitrary files, which do not contain any change history (i.e. each file has only a single revision) or only changes that affect whole file contents. In this case, our encryption scheme would generate chunking-related metadata overhead that is not compensated by space savings due to similar file revisions.

To verify whether these drawbacks are of practical relevance, we evaluated our solution using some real-life data. For this, we re-enacted the version history of the trunk of the open source project *ispCP*[11], which consists of many small (≈ 10 KiB) source code files as well as rarely-changed files such as pictures. Thus, this example combines elements from both problematic scenarios discussed above. Analogously to the previous experiments, the results are shown in Fig. 2. Since the repository starts with a kind of worst-case situation (at the beginning, 2 768 small (≈ 10 KiB) files are added), the storage efficiency of our solution seems to be worse than the one with traditional encryption. With an increasing number of revisions, however, our solution's savings compared to traditional encryption get significant again.

[11] `http://isp-control.net`; repository: `http://www.isp-control.net:800/ispcp_svn/trunk`, requested on 2012-03-26

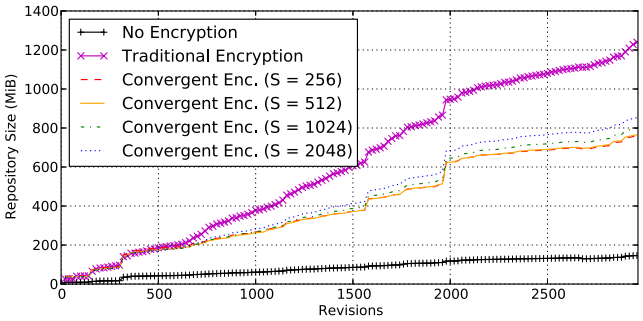

Figure 2: Development of ispCP's repository size

A.3 Memory and Time Requirements

We have seen that our access control solution allows efficient storage of encrypted repositories. We now consider the amount of memory and time needed to en-/decrypt confidential files. Besides the specific implementation, there are conceptual aspects critical to memory / time consumption. A trivial implementation has two main drawbacks:

1. Encryption consumes memory in the order of about 48 times the file size as it has to count the appearances of each 48-byte window content (see Sec. 5.1.1).
2. As HMAC computations are time-consuming, computing a rolling HMAC (one computation for every byte of the file size) significantly slows down encryption.

To find a suitable trade-off between memory / time efficiency and security, we evaluated 4 variants of our algorithm:

- *HMAC, no repetitions*: This is a trivial reference implementation of our concept, which implements appearance counters using hash tables. This version is memory-consuming, but adheres strictly to the description in Sec. 5.
- *HMAC, no repetitions, bloom filter*: This variant is similar to the first, but implements the rolling hash appearance counter using a bloom filter [2], achieving memory consumption of about 3 times of the file size by allowing *false positives* (i.e. repetitions could be detected by mistake). As false *negatives* are excluded, this only affects storage efficiency, not security (rolling hash values might change, but repetitions are still excluded).
- *HMAC*: This variant ignores repetitions of rolling hash values, so periodical file contents could lead to periodical chunk boundaries visible in ciphertexts. This has a slightly negative effect on the security properties when applied to files with repeating contents, but—if this limitation is acceptable—significantly reduces memory consumption.
- *Rabin fingerprints*: By using an efficient rolling hash function (i.e. *Rabin fingerprints* [16]) for rolling hash computation (and still ignoring repetitions), encryption is speeded up dramatically. However, while we expect its impact on security to be negligible, we do not recommend this variant as it might leak information about chunk boundaries.

The experiments—performed on an *Intel(R) Core(TM) i5-2500K* machine with 16 GiB RAM—confirmed our expectations: With 3.3 seconds for encrypting a 16 MiB file, the Rabin fingerprint variant performs about 10 times faster than the corresponding HMAC variant—but leads to security drawbacks. The significant decrease (factor 10) of memory consumption achieved by the bloom filter is also done at the expense of computing time (factor 2). When considering only the two provably secure variants, we consider the bloom filter variant the best trade-off. Thus, this variant is our default and has been used for the other experiments.