A Trusted Versioning File System for Passive Mobile Storage Devices

Luigi Catuogno[a], Hans Löhr[b,c], Marcel Winandy[b], Ahmad-Reza Sadeghi[d]

[a]*Dipartimento di Informatica*
Università degli Studi di Salerno
Via Ponte Don Melillo
I-84084 Fisciano SA - Italy

[b]*System Security Lab*
Ruhr-University Bochum
Universitätsstraße 150
D-44801 Bochum - Germany

[c]*Now working for*
Robert Bosch GmbH, Corporate Research
D-70442 Stuttgart - Germany

[d]*System Security Lab*
Technical University Darmstadt
Mornewegstraße, 32
D-64293 Darmstadt - Germany

Abstract

Versioning file systems are useful in applications like post-intrusion file system analysis, or reliable file retention and retrievability as required by legal regulations for sensitive data management. Secure versioning file systems provide essential security functionalities such as data integrity, data confidentiality, access control, and verifiable audit trails. However, these tools build on top of centralized data repositories operating within a trusted infrastructure. They often fail to offer the same security properties when applied to repositories lying on decentralized, portable storage devices like USB flash drives and memory chip cards. The reason is that portable storage devices are usually passive, i.e., they cannot enforce any security policy on their own. Instead, they can be plugged in any (untrusted) platform which may not correctly maintain or intentionally corrupt the versioning information on the device. However, we point out that analogous concerns are also raised in those scenarios in which data repositories are hosted by outsourced cloud-based storage services whose providers might not satisfy certain security requirements.

In this paper we present TVFS: a Trusted Versioning File System which stores data on untrusted storage devices. TVFS has the following features: (1) file integrity and confidentiality; (2) trustworthy data retention and retrievability; and (3) verifiable history of changes in a seamless interval of time. With TVFS any unauthorized data change or corruption (possibly resulting from being connected to an untrusted platform) can be detected when it is connected to a legitimate trusted platform again. We present a prototype implementation and discuss its performance and security properties. We highlight that TVFS could fit those scenarios where different stakeholders concurrently access and update shared data, such as financial and e-health multiparty services as well as civil protection application systems such as hazardous waste tracking systems, where the ability to reliably keep track of documents history is a strong (or legally enforced) requirement.

1. Introduction

Versioning file systems transparently retain different versions of stored files and record the history of changes made to each file over time. This kind of file systems was originally introduced to allow users to maintain a file-grained backup making available an accurate log of their work and possibly to recover their data after any wrongful operation. Versioning file systems have several advantages compared to conventional backup systems or application-level revision control systems [1]. Versioning file systems store locally the old data, so that users can autonomously access their backups without the effort of the system administrator; they operate independently from the type of files and the applications; new versions of any file are created automatically each time a data change occurs, instead of requiring the user who modifies the file to explicitly create a new version.

Email addresses: luicat@dia.unisa.it (Luigi Catuogno), mail@hans-loehr.de (Hans Löhr), marcel.winandy@trust.rub.de (Marcel Winandy), ahmad.sadeghi@trust.cased.de (Ahmad-Reza Sadeghi)

These characteristics make versioning file systems suitable to several important applications, motivating renewed interest in this subject. For example, running an operating system over a versioning file system, enables the administrator to detect, and analyze the occurring data changes in order to discover potential malicious activities and, if any, provides the capability to recover the file system to a safe state. Furthermore, versioning file systems naturally fit those application scenarios where, due to recently introduced regulations – *e.g.*, Sarbanes-Oxley Act (SOX), Health Insurance Portability and Accountability Act (HIPAA) and the Federal Information Security Management Act (FISMA), – retention and availability of sensitive data, as well as reliable information about their evolution, are recommended or mandatory.

Therefore, newly introduced *secure* versioning file systems [2, 3, 4, 5] feature trustworthy data repositories which guarantee confidentiality, integrity and availability for each stored file version, according to the required security and retention policies. In addition, any change made to the stored files is recorded to accurate log files in order to keep verifiable tracks of which data have been modified, when and by whom.

Generally, these file systems provide a data repository which is embedded within a trusted computing base that ensures the file contents and meta-data are protected and can be accessed only through a precise protocol and according the configured security policy. Such repository can be either implemented locally and managed by a trusted operating system component, or can be hosted by a remote centralized file server, and reachable through the network, leveraging on network security facilities as communication encryption and peers authentication.

However, we highlight that such tools cannot provide equivalent guarantees if the data repository is implemented on a Mobile Storage Device (MSD). Examples of MSDs are USB memory sticks, transportable hard-disk drives, and memory chip cards. MSDs have gained an important role as means to easily transfer data across multiple working locations and platforms, and to store and transfer data among equipment of very different type, *e.g.*, digital cameras, printers, car or home media systems, and even medical equipment. A typical usage scenario are medical alert USB flash drives that store important medical information of users on the stick to be used in emergency situations, such as accidents or natural disasters.

However, introducing the possibility of storing data on MSDs raises serious security concerns. In general, the usage of passive mobile storage devices faces the challenges of protecting the confidentiality of the stored data and protecting the computing platforms they are attached to from potential malware that may reside on the storage devices. In particular, if used as versioning data repository, mobile storage devices allow to perform unattended changes to the stored files, *i.e.*, bypassing the correct protocol and security policy enforcement. The reason is that they are passive storage devices, they generally do not have an implicit security mechanism built in. Hence, passive MSDs are totally under the control of the platform they are attached to. Moreover, standard encryption tools, even when correctly executed on those computing platforms, may solve the confidentiality requirements, but they usually lack support for maintaining versioning information and audit trails on the storage device.

We mention in the following paragraphs two real-world scenarios related to civil protection services that face these issue in practice.

Nowadays electronic healthcare infrastructures provide citizens of personal storage devices which store some essential medical information, as well as maintain individual collections of electronic health information: e.g., the electronic health record (EHR), potentially shareable across different domains (such as, practitioners, social security or insurance officers, and so on) with different access privileges. All these stakeholders may access and update this information according to their roles and satisfying certain access control and privacy rules. However, any data access and manipulation should be traceable and possibly reversible, as erroneous, fraudulent, and no longer valid changes should be detectable and revocable as well. However, several e-health infrastructure implementations provide such mechanisms in order to guarantee the security and privacy of data lying on centralized storage repository [6], whereas enforcing the same security policies over the data stored on the individual personal devices still raises several issues [7]. Consider the case of a car accident, first-aid personnel needs to be promptly enabled to access the data stored in the victims' personal device. To this end, any required access credential is disclosed, bypassed any access control rule. The main drawbacks of this procedure are that data become completely prone to potentially destructive operations, sensitive information, though not related to the current emergency, could be disclosed, revoking such emergency-driven access privileges may be not a trivial task.

Waste management systems include monitoring of waste disposal processes with particular emphasis in tracking the transfer of hazardous waste and maintainig the related information. A typical example is the (now abandoned) italian hazardous waste tracking system (SISTRI) [8] which aimed to provide an electronic framework to track the waste chain at a national level. It featured several devices, including a passive "trusted" MSD (a USB stick) which stored several documents such as waste delivery notes and packing slips. The waste producer created the related documents (with a trusted application also lying on the MSD) and gave it to the transporter who plugged it in a "black-box", a trusted device mounted on the vehicle which enriched the documentation with annotations about the trip (e.g., GPS measures, delivery to different transporters or brokers) to the stocking site. One of the issues raised by this system was ensuring the reliability of the annotations history.

In this paper, we concentrate on solving the problem of implementing secure file-grained versioning upon a data repository lying on a totally passive off-the-shelf storage device, such as Mobile Storage Devices, aiming at preserving as much as possible of their flexibility which is probably the main reason of their success. To this end, we present the design and the implementation of a Trusted Versioning File System (TVFS) which features file integrity and confidentiality along with a verifiable log of changes mechanism to guarantee that: opening a file ensures that it was created or modified only by the set of users resulting from its history, and that all versions one can extract from the file define its history of changes in a seamless interval of time.

In contrast to existing file systems or versioning systems, TVFS enables two major benefits:

- TVFS' security properties do not rely on any security feature provided by the underlying storage device. It allows to store and reliably retrieve files and their related versioning information even on mobile storage devices that can be moved among computing platforms *without the need for a central repository system.*
- TVFS provides a security model that affects only slightly the way in which MSDs are traditionally used: It does not impose neither additional security procedures, nor further constrains to frequent operations such as making unattended device backups or multiple copies of versioning files.

TVFS leverages on the Content Extraction Signature (CES) scheme [9, 10, 11]. As far as we know, this is one of the earliest applications of this cryptographic primitive in file systems design.

Paper organization. The paper is organized as follows. Some preliminary notions and notation about Content Extraction Signature and versioning file systems are given in Section 2. Our proposal is introduced in Section 3, whereas details of the underlying file format and operation are treated respectively in Sections 4 and 5. The security of the scheme is analyzed in Section 6. A proof-of-concept implementation along with its performance are discussed in Section 7. The paper provides a brief survey of proposals and research related to our work in Section 8 and some concluding remarks in Section 9.

2. Background

2.1. Content Extraction Signature

Our file system uses the Content Extraction Signature (CES) scheme proposed in [9, 10, 11]. In this scheme we have three actors: the signer $\mathcal{S}$, the bearer $\mathcal{B}$ and the verifier $\mathcal{V}$. $\mathcal{S}$ produces and signs a document and gives it to $\mathcal{B}$.

$\mathcal{B}$ can extract a subdocument (i.e., a portion of the original document) along with an *extracted signature* without any interaction with $\mathcal{S}$ and without knowing her private key. However, while signing the document, $\mathcal{S}$ specifies which parts of it $\mathcal{B}$ is allowed to extract a valid signature for. Eventually, $\mathcal{B}$ sends her subdocument to $\mathcal{V}$ who can verify that the subdocument has been signed by $\mathcal{S}$ without any interaction with her and without knowledge of the portion of the original document $\mathcal{B}$ did not include in the issued subdocument.

We use this scheme to implement trustworthy traceability and reversibility of file content changes.

2.1.1. Notation and Definitions

A *document* M is composed of an ordered set of n smaller *messages* $M = \{m_1, m_2, m_3, \ldots, m_n\}$. An extracted subdocument M' of M is composed of a subset of messages of M so that the i-th message of M' can be either m_i or $\flat$ (blank/empty) while the position of the other non-blank messages is the same as in M. The i-th message of document M is denoted as $M[i]$. The set $\{1, \ldots, n\}$ of message indices in M is denoted with $[n]$. The *extraction subset* X is the set of the indices of messages of a document M that are included in a subdocument M'. X is a subset of the set of message indices in M. Given a document M, the *Content Extraction Access Structure* (CEAS) C represents the set of all allowed extraction subsets, as it is stated by the signer $\mathcal{S}$.

Given a document $M = \{m_1, m_2, m_3, \ldots, m_n\}$, any document N composed of the same messages of M placed in a different order, is considered a different document (*e.g.*, $N = \{m_2, m_1, m_3, \ldots, m_n\} \neq M$).

Example. Let $M = \{m_1, m_2, m_3, m_4\}$, the document $M' = \{m_1, \flat, m_3, m_4\}$ and $M'' = \{\flat, m_2, \flat, m_4\}$ are subdocuments of M whereas $Q = \{m_1, m_3, m_4\}$ and $Q' = \{m_2, m_4\}$ are not.

Moreover, $X' = \{1, 3, 4\}$ and $X'' = \{2, 4\}$ are, respectively, the *extraction subset* of M' and M''.

Definition. A Content Extraction Signature (CES) scheme as defined in [9, 11] consists of four algorithms:

- `Keygen()`. On input of a security parameter κ, generates a secret/public key pair (SK, PK)
- `Sign()`. On input of the signing key SK, a document $M = \{m_1, \ldots, m_n\}$ and a content extraction access structure C, outputs a content extraction signature σ_M
- `Extract()`. On input of a document M, its σ_M, the signer's public key PK and an extraction subset $X \subseteq [n]$, outputs an extracted signature σ_{M_X} on the subdocument M_X.
- `Verify()`. On input of an extracted subdocument M_X, its σ_{M_X} and PK, outputs a verification decision $\in \{accept, reject\}$.

2.1.2. Properties

An essential property a Content Extraction Signature scheme is required to satisfy is the following:

Definition. **CES-Unforgeability** It is infeasible for an attacker, having access to a CES signing oracle, to produce a document/signature pair (M, σ) such that: σ passes the verification test for M and M is either: (a) not a subdocument of any document queried to the CES signing oracle, or (b) is a subdocument of a queried document D, but not allowed to be extracted by the CEAS attached to the sign query for D.

All schemes presented in [9, 10], are proven to be *CES-unforgeable* if the standard signature scheme $\mathcal{S}$ they are built upon, satisfies the standard unforgeability notion, *i.e.*, it is existentially unforgeable under adaptive chosen message attacks (see [12]).

In particular, our file system leverages on the instantiation proposed in [11], which is proven to be CES-unforgeable.

2.2. Versioning File Systems

Versioning is a technique that allows to record files along with their history (*i.e.*, the sequence of changes they have been subject to, throughout their lifetime), so that one can reconstruct any past version of the file, in order to recover possible wrongful changes, as well as to realize how any file has changed over time. This technique can be implemented in different ways, according to the application field, for example, revision control systems, such as RCS [13] or CVS [1] are consolidated versioning tools that are available free of charge and mainly employed in software development.

In general, files and history-related metadata are stored in a file repository which is exclusively maintained by the versioning system and that can be accessed only through a certain protocol, implemented by the provided versioning tools.

Existing versioning systems, provide a plethora of common basic functionalities, we mention in particular:

- **Check-in**: is the operation through with a new file version, currently present in a temporary working area, is added to the repository.

- The **check-out** procedure retrieves an arbitrary version from the repository and materializes it into a temporary working area. Usually, the check-out operator returns the latest version in the repository. However, one can *select* and check-out a different version, by explicitly indicating its version number or tag.

- **Version merging and pruning**. The version number tends to grow indefinitely, affecting the system's performance. These operators allow to reduce it, by deleting useless (or wrongful) versions as well as merge together different consecutive versions which carry little changes each.

- **Comparing versions**. Evaluating how stored files change is a central feature in a versioning system. Typically, a versioning system should enable its users to print out the differences between two versions or between the selected version and the one present in the workspace.

- **Attribute management**. Attributes are information expressed as couples {*attr-name,attr-value*} that can be associated to each version. These attributes can be used by the versioning system itself as well as by applications which leverage on it.

- **History analysis** includes a set operations that are used to formulate queries to the repository. Very often, users may want list the file's history, according certain lemmas or aggregate actions according certain version attributes or properties (*e.g.*, author, timestamp, annotations).

A versioning file system is essentially a revision control system that exposes the interface of traditional file systems. On a versioning file system, the user applications access implicitly to the current file version, whereas, a new version is silently created whenever the file content is modified by means of write operations. Generally, applications are unaware of accessing versioned files as the system is implemented as a virtual file system layer overlaying the real file repository.

Although main file operators can be mapped onto versioning operators, not every versioning functionality can be mapped onto the file system semantic. For example, functionalities such as retrieving an older file version or browsing the change history of a file should be performed by means a separated interface, e.g., a set of ad-hoc version management tools. In Section 8, we briefly survey the literature related to versioning file systems.

2.3. Lazy revocation

Lazy revocation [14] is a technique used to efficiently handle user revocation in groups of users which share encrypted resources and the related encryption key. A lazy revocation scheme assumes that protecting old data from revoked users is not necessary since they could have already accessed the data and have potentially disclosed it. Hence, when a user is revoked, a new key is issued and delivered to the remaining group members, but previously encrypted data are not re-encrypted, whereas new data will be encrypted with the new key. Note that each user still needs to store the old keys in order to read data encrypted at the respective time of validity.

More precisely, consider a group of users who share some encrypted data. In a lazy revocation scheme, each key is assigned of a validity interval of time (time-slot). So, if t is the current time-slot, k_t is the currently valid key, and all keys k_i generated at times $i < t$ are considered revoked. Whenever a user leaves the group, the current key is revoked and the new key k_{t+1} is generated and delivered to the remaining group members. To avoid that participants store all revoked keys, several schemes [15, 16] provide users of a single *user master key* K_t for each time-slot t. K_t can be used to *extract* all keys k_i $(0 \leq i \leq t)$. This kind of scheme is characterized by a *trusted status* for each time-slot t. The initialization algorithm of the lazy revocation scheme generates the initial engine state E_0 related to the time-slot $t = 0$. User master key K_0 is *derived* from E_0. When a revocation occurs, the scheme *updates* its state taking current state E_t to the new state E_{t+1}, hence, a new master key K_{t+1} is derived and delivered. Revoked users still know K_t, but cannot use it to *extract* the new key k_{t+1}.

3. Our proposal

In this section we present the Trusted Versioning File System (TVFS), our solution to the problem of implementing a trustworthy versioning file system upon totally untrusted storage, such as mobile storage devices. We first give a general overview of the file system design and basic assumptions before we focus on the details in the subsequent sections.

TVFS features a file system layer that provides users a standard file system interface to the underlying versioning repository. TVFS implements and enriches the standard file system calls with the required version management operators such as *check-in* and *check-out*.

For example, when a process opens a file with the `open` system call on a TVFS volume, the TVFS implementation performs a *check-out* operation to the underlying repository. Subsequently, a working copy of the requested file *as it appears* in its latest version is made available in a temporary storage area. Read and write operations are done on this working copy through a transparent encryption layer, achieving confidentiality of the content. The `close` system call in turn leads to the *check-in* of the currently modified file version and destroys the working copy. Figure 1 shows an overview of the TVFS file system layer operation.

A set of additional versioning tools implements the repository management operations that do not fit the usual file system semantics. This includes functions to check-out a version that is different from the latest, to compare and extract differences between two versions, and to show and analyze file history annotations.

The different versions of each file are stored in a special data format. This data format contains: (i) the latest checked-in version of the file in encrypted form; (ii) all changes (encrypted) made to the file in order to produce it since file creation; (iii) meta-data to reconstruct every single file version and to identify the corresponding author thereof; and (iv) for each version, the extracted signatures that allow to verify whether the version is trusted and has been built starting from a trusted version.

The file repository is, in turn, implemented by means of standard file system calls, so that it is completely independent from the underlying storage device as well as its running file system type. Indeed, a TVFS repository can be equally stored on a NTFS formatted portable disk or a FAT32 formatted USB stick.

A TVFS volume is assumed to operate within a security domain encompassing a Public Key Infrastructure, which features a certificate authority devoted to issue and revoke certificates for signature keys, and a generic *credential authority* which takes care of encryption key management and access control policy enforcement.

Within such a domain, legitimate users access any TVFS file system through a set of trusted platforms that:

- establish a secure connection with the credential authority in order to retrieve the user credentials and keys needed to properly access and verify stored files, as well as to securely store such credentials locally, protecting them from unauthorized accesses and misuses;
- locally enforce the access control policy and guarantee that the repository is accessed only following the right protocol, that is, ensuring that any update to the file repository is done respecting syntactically and semantically the TVFS file format.

As TVFS repositories are stored on totally passive storage devices, such as dumb flash drives, we must expect them to be mounted to, accessed and modified by, as well as copied and variously manipulated by any platform they are attached to. However, under the requirement stated above, any unauthorized data change or corruption can be immediately detected as the device is connected to a legitimate platform.

We focus on the implementation and evaluation of the file system layer in this paper, whereas we leave the credential authority and other components of the security domain at a more abstract level. Since pursuing a complete separation and independence between the file system design and key/user management functionalities, we envision that these aspects can be instantiated by many existing solutions. In particular, in developing TVFS we have considered as reference architectures those described in [17] and [18].

3.1. Preliminaries and notation

In TVFS, any file F is shared by a group of users $G = \{u_1, \ldots\}$. The credential authority is the component that performs any change to the group configuration, e.g., adds new users to the group, and revokes existing group members. Whenever a new user joins the group, its newly created identifier u is added to G, whereas the identifier of any revoked user u_i is added to the set of revoked users $RL = \{u_{r_1}, \ldots\}$. Each user $u_i \in G$ has an own public/secret key pair (PK_{u_i}, SK_{u_i}) used to sign and verify files shared with members of G.

Our scheme features file content encryption by means of a symmetric block cipher (see Section 5.4). The set

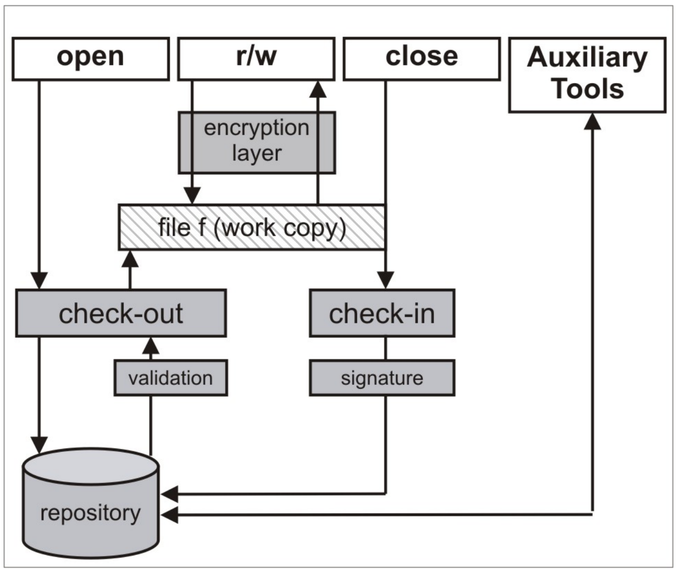


Figure 1: System Operation

$K = \{k_1, \ldots\}$ contains the keys to encrypt files shared by members of group G. The credential authority handles keys in K according to the file access policy agreed among group members and following the usual key lifecycle events (creation, revocation, renewal due to user revocation, etc.). We denote the configuration of group G with the tuple $< G, RL, K >$. Group members access the shared files through their *user platform* (UP). For simplicity, we assume user platforms are *individual*, so that user u_i accesses shared files through platform UP_i.

Any TVFS file F is stored as the ordered sequence $\{V_0, \ldots, V_l\}$ of records that represent all its version from its initial version V_0 to the latest one V_l (see Section 4). Each version V_i has been produced consequently to the changes made to F by a certain group member $u_j \in G$. Once users have created a new version V_i, they also sign the file — we describe the signature scheme in Section 5. We say that user u_j is the *signer*, denoted as s_i, of version V_i. Note that, for any choice of versions V_i and $V_k \in F, (i \neq k)$, it could be that $s_i = s_k = u_j$ where u_j is a certain member of G who created and signed both versions.

The solution we present guarantees:

- that F preserve its integrity and confidentiality;
- that F has been created by user s_0 and subsequently modified by users $s_1, \ldots, s_l$; and
- that the ordered set of versions $\{V_0, \ldots, V_l\}$ exactly defines the history of F in a seamless time interval.

4. File structure

A shared versioned file F is stored as an ordered sequence of file versions $V_0, \ldots, V_l$, where V_0 is the *initial* version and V_l is the *latest* one. The initial version V_0 is created along with the file and does not contain user data.

A file version V_i, $i > 0$ is composed of a set of blocks $\{h_i, b_1, \ldots, b_n\}$, the special block h_i is the *header block* of version V_i. In our application, each file version plays the role of a document M in the content extraction signature given above. We denote the j-th block in V_i with $V_i[j] = b_j$. In case of ambiguity, we denote the i-th version of file F as $F.V_i$.

Blocks in a file version V_i can be either *old* or *new*. Old blocks come unchanged from the previous file version V_{i-1} whereas new blocks have been changed in the current version. Furthermore, we denote as *undo* blocks, those blocks of V_{i-1} that have been replaced by new blocks in V_i.

As in most versioning systems, we adopt the strategy of keeping the current version of a shared file entirely available, whereas the previous versions are stored as *undo records*, i.e., as the patch that has to be applied to the latest version in order to obtain the previous one. In our file format, the undo record representing any intermediary version V_i is composed of the undo blocks of version V_{i+1}.

Moreover we have to distinguish between *latest* and *current* versions. The current version is the one that is currently accessed by the user. By default, the current version is also the latest but this may not be true if, for example, the user wishes to work on an older version in order to start a different branch of the file. In this case, before allowing the user to access the requested version, the system first reconstruct and selects it as current. We denote the current version with V_c. Trivially, if V_l is the latest version and V_c is the current version, $c \leq l$.

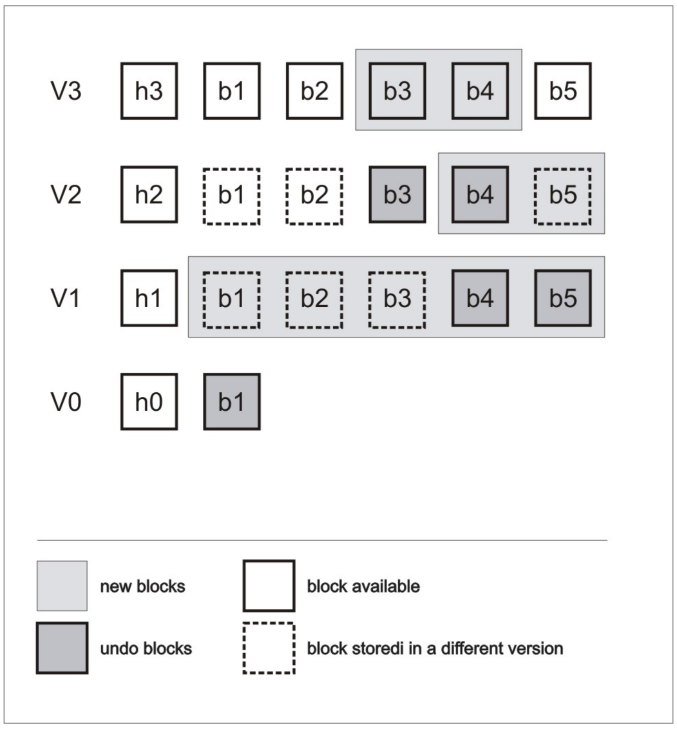


Figure 2: Sample file with three versions.

Let's consider the example depicted in Figure 2. The file F is composed of three versions (we do not count the initial version V_0) represented by the set $V_0, \ldots, V_3$. Version V_3 is composed of the blocks $\{h_3, b_1, b_2, b'_3, b'_4, b_5\}$ where h_3, b'_3 and b'_4 are new blocks, b_1, b_2 and b_5 are old blocks and, consequently, the undo record representing V_2 is composed of block h_2 and previous version of blocks b_3 and b_4.

We define as *old(V_i)* the extracted subdocument of V_{i-1} containing h_{i-1} and all old blocks in V_i, and as *undo(V_i)* the extracted subdocument of V_{i-1} containing all undoblocks of V_i. In our example *old(V_3)*$=\{h_2, b_1, b_2, \flat, \flat\, b_5\}$ and *undo(V_3)* $= \{h_2, \flat, \flat, b_3, b_4, \flat\}$.

The header block contains the information needed to reconstruct and validate the whole file version. More in detail, the header block h_i of a certain version V_i contains: the identity of the signer s_i, and three signatures: $\sigma^{(i)}_{full}$, $\sigma^{(i)}_{old}$ and $\sigma^{(i)}_{undo}$ that are respectively: the signature of the whole i-th version computed by s_i and the extracted signatures of the subdocuments *old(V_i)* and *undo(V_i)* of V_{i-1}.

Further file metadata is also placed in the header block, such as file length, the number of blocks, information about data encryption, etc.

Note that, since it changes at each version, the header block is always present in each undo record. Moreover, for each version V_i the header block h_{i-1} also belongs to *old(V_i)*, though it never appears within the old blocks of any version. This is because h_{i-1} is needed to verify the extracted signature of *old(V_i)*.

The content of a File version is encrypted on a per block basis. Each block can be encrypted, with a different key belonging to the group key set K. For example, given a version $V_i = \{h_i, b_1, b_2, \ldots\}$ the identifier of the key used to encrypt any block b_j is stored as k_j in a meta-data section of the block data structure.

5. File operation

5.1. Creating and revising versioned files

We describe how new files and new versions of an existing file are created. In the following we will refer to a file $F = \{V_0, \ldots, V_l\}$ and denote as s_i the signer of $V_i, (0 \le i \le l)$ (note that for some $i \neq j$ can be $s_i = s_j$).

When s_0 creates the new file F, the file system creates its initial version V_0. Such a version contains only the header block h_0.

The first version V_1 is separately created when the earliest data is written to the file.

Here we describe this process along with the procedure followed to create a new version to an existing file. We recall that $V_i[j]$ denotes the $j-th$ block b_j of version V_i and, in particular, $V_i[0]$ denotes the header block h_i.

Let $V_l = \{h_l, b_1, b_2, \ldots, b_n\}$ be the latest version of file F. Assume user s_{l+1} opens F and modifies a set of blocks in V_l whose indices form the set J (note that always $0 \in J$). The file system creates the new version V_{l+1} whose blocks are: $V_{l+1}[j] = b'_j$ if $j \in J$ and $V_{l+1}[j] = V_l[j]$ otherwise.

For each version, the file system computes two signature extractions and one new full signature. These signatures are stored in the header block h_{l+1}, and are computed as follows:

- $\sigma^{(l+1)}_{old} = \texttt{Extract}(V_l, \sigma^{(l)}_{full}, PK_{s_l}, [n] - J \cup \{0\})$ is the extracted signature of the old blocks of V_{l+1} and assures that such blocks come from V_l (without reconstructing V_l and without contacting u_l).
- $\sigma^{(l+1)}_{undo} = \texttt{Extract}(V_l, \sigma^{(l)}_{full}, PK_{s_l}, J)$ is the extracted signature of the undo blocks of V_{l+1} and allows to verify that such blocks come from version V_l.

For each version V_i, we denote with $\mathcal{C} = \{J \in 2^{[n]} | 0 \in J\}$ the default CEAS. This setting states that all extracted subdocument of V_i include the header block h_i.

Then, the system computes the signature of the whole version V_{l+1} as $\sigma^{(l+1)}_{full} = \texttt{Sign}(SK_{s_{l+1}}, V_{l+1}, \mathcal{C})$, and adds it to h_{l+1}. On the filestore, the previous version V_l is replaced by its subdocument *undo*(V_{l+1}) and eventually, the entire new version V_{l+1} (e.g., all its blocks) is appended to the file F.

5.2. Version extraction and verification

As we said above, in a versioned file $F = \{V_0, \ldots, V_l\}$, each version $V_i (0 < i \le l)$ is formed by adding just new/modified blocks and keeping all unchanged blocks of V_{i-1}. On the other hand, except the latest version V_l which is always completely available, any intermediate version $V_i, (0 < i < l)$ is reconstructed (on demand) by applying iteratively to V_l, all undo records representing versions $V_j, (i \le j < l)$.

The full signature of the current version V_i along with the identifier of its signer s_i are retrieved from its header block h_i. If V_i is not the initial version, h_i also contains the extracted signatures of old and undo blocks whereas the signer identifier s_{i-1} is available in the header block h_{i-1} (that is stored in the undo record that represents that previous version.)

Let u be a user who checks out the version V_i from F. User u can check-out that version if she is able to reconstruct and successfully verify the integrity of all $V_j (i < j \le l)$ (no matter if any signer $s_j \in RL$). Moreover, u can trust that version if she can *validate* it, i.e., if she can successfully verify the integrity of V_i and all previous versions $V_j, (0 \le j < i)$ and, moreover, not one of the users that signed those version appears in RL.

More precisely, we say that a version V_i is *valid* if:

- its full signature $\sigma^{(i)}_{full}$ is successfully verified and was generated by a non-revoked user s_l, i.e., $\texttt{Verify}(PK_{s_i}, V_i, \sigma^{(i)}_{full})$ is *true* and $s_l \notin RL$.
- for $i > 0$, the extracted signature $\sigma^{(i)}_{old}$ of old blocks of V_i is successfully verified, that is: $\texttt{Verify}(PK_{s_{i-1}}, old(V_i), \sigma^{(i-1)}_{old})$ is *true.*
- for $i > 0$, the extracted signature $\sigma^{(i)}_{undo}$ of undo blocks of V_i is successfully verified, i.e., $\texttt{Verify}(PK_{s_{i-1}}, undo(V_i), \sigma^{(i-1)}_{undo})$ is *true.*

```
VERIFY-VERSION($F, V_c$):
    $res = \texttt{Verify}(PK_{s_c}, V_c, \sigma^{(c)}_{full})$
    if $c = 0$:
        return $res$
    return $res \wedge$
            $\texttt{Verify}(PK_{s_{c-1}}, old(V_c), \sigma^{(c)}_{old}) \wedge$
            $\texttt{Verify}(PK_{s_{c-1}}, undo(V_c), \sigma^{(c)}_{undo})$


RECONSTRUCT($F, V_c, r$):
    if $c = r$:
        return $V_c$
    let $J_c = \{j | undo(V_c)[j] \neq \flat\}$
    build $V_{c-1}$ such that:
        $V_{c-1}[j] = \begin{cases} undo(V_c)[j] & \text{if } j \in J_c \\ V_c[j] & \text{otherwise} \end{cases}$

    if VERIFY-VERSION($F, V_{c-1}$) is True:
        return RECONSTRUCT($F, V_{c-1}, r$)
    else return $\perp$


VALIDATE($F, V_c, lsv$)
    if $c = 0$:
        if $s_c \in RL$:
            return $\perp$
        else return $lsv$

    $V_{c-1} \leftarrow$ RECONSTRUCT($F, V_c, c-1$)
    if $V_{c-1}$ is $\perp$:
        return $\perp$

    if $s_c \in RL$:
        $lsv \leftarrow V_{c-1}$
    else $lsv \leftarrow latest(V_c, lsv)$

    return VALIDATE($F, V_{c-1}, lsv$)
```

Figure 3: Pseudo-code of algorithms RECONSTRUCT and VALIDATE

In Figure 3 we show a description of algorithms RECONSTRUCT() and VALIDATE().

5.3. Handling Invalid File Versions

Now let us explain how the validation process allows the file system to handle non-valid file versions.

We highlight that signature verification failures and "revoked signatures" are treated differently. Indeed, when a signature verification fails, it is not possible to establish whether the version which caused the failure has been altered rather than its signature. Moreover, if such a version shares some blocks with other versions, it is not possible to establish which is the bad one. Therefore, whenever such events occur, the validation process fails and the file is considered corrupted.

However, we assume that if revoked users still created consistent versions, although the content might be somehow "wrong" or "malicious" (which, for instance, might have been the reason for revocation), their versions can be detected and the changes they introduce can be recovered.

Therefore, when the checkout of a certain version V_i is requested, there are three possible results:

- The requested version V_i is successfully built and validated.
- The validation of version V_i failed. Version $V_k \neq V_i$ and $0 \leq k < i$ (where k is the highest value such that V_k is valid) is returned instead. This means that all versions $V_j, (k < j \leq i)$ are not valid, because of signer $s_j \in RL$.
- $V_j = \perp$ (failure) If during the validation of any version V_i of F a signature verification failure occurs, the whole file is considered non-valid, the validation process immediately ends and an error message is notified to the process that was accessing the file.

5.4. File Encryption

In our scheme, file encryption is performed by means of a lazy revocation scheme. This choice has one important reason: re-encrypting every involved files, whenever a key revocation occurs, is not possible since Mobile Storage Devices on which those files lie, may be not available.

5.4.1. Encryption Key Management

As introduced above, members of group G agree on a set of cryptographic keys $K = \{k_1, \ldots\}$ in order to encrypt (by means of a symmetric algorithm) the content of any shared versioned file. We denote with t the time of validity of the key $k_t \in K$. At a certain time $t = 1$, when the group G is formed, members use an initial set of keys $K = \{k_1\}$ containing just the first encryption key. Afterwards, whenever one of the usual events in the group's life-cycle occurs (e.g., a user leaves the group, the encryption key is disclosed), the GA revokes the key currently in use k_t, generates a new key k_{t+1} and adds it to the set K. So that, at time $t + 1$, we have $K = \{k_1, \ldots, k_t, k_{t+1}\}$, k_{t+1} is the *current key* and any other key $k_i \in K$ where $(1 \leq i < t+1)$ is revoked.

5.4.2. Revising Files

Let $F = (V_0, \ldots, V_{n-1})$ be a versioned file shared by group G, and let $u_i \in G$ be the user who adds the new version V_n to F at time t. New blocks are mandatorily encrypted using the current key $k_t \in K$ and added to the file according to the scheme described above. The key-id field of each block $b_i \in new(V_n)$ is set to k_t whereas blocks in $old(V_n)$ and $undo(V_n)$ remain unchanged.

5.4.3. Reading Files

Reading operations over the versioned file F may involve blocks created in different time slots and hence, encrypted with different encryption keys. For example, let the user u_i, who accesses the file F from an MSD attached to the platform UP_i, read the sequence of m blocks $\{b_{j_1}, b_{j_2}, \ldots, b_{j_m}\}$. In order to decrypt the block's content the user u_i retrieves the keys $\{k_{j_1}, \cdots, k_{j_m}\}$ from the set K.

5.4.4. Dealing with missing encryption keys

If UP_i is an off-line platform, it may happen that some blocks of the version V_n of file F have been encrypted with a key that is not present in K. This is the case in which user u_i has been revoked, thus, the platform is not allowed to handle the file version which contains that block whereas is still enabled to access to previous versions of F. Therefore, the platform is forced to search for and to check out the latest file version (say $V_k, k < n$) in which the most recent encryption key is still present in K. Once that version (if any) is reconstructed, the platform can perform read operations as usual, whereas write operations will lead to the creation of a new *branch* of F. In our scheme, this operation creates a new file F' whose version V'_1 is the copy of version $V_k \in F$ and any possible change made by u_i is stored in the following versions of F'.

6. Security Analysis

In this section we discuss the security model of TVFS. In particular, we consider two important aspects:

1. Any TVFS file can be freely replicated, even by untrusted users and platforms. Each unmodified copy is considered valid while it can be successfully verified.
2. Each copy F' of $F = (V_0, \dots, V_l)$ is considered a valid branch of F. In particular, if a non-revoked user u_k legitimately creates a new file $F' = (V'_0, \dots, V'_k)$ where $(V'_i = V_i, 0 \leq i \leq k)$ and possibily new valid versions $V'_{k+1}, V'_{k+2}, \dots$ are added, the resulting file F' is considered a valid branch of F if V'_k is a valid version of F. Moreover, file F' is still a valid branch of F, even if any user u_j who signed a version $V_j \in F, (j > k)$ has been revoked.

In the following, we describe the main security threats a versioning file system may encounter in our application scenario, and we argue how TVFS copes with them.

6.1. Threat model

In our model, we consider insider attacks targeted at changing the file history or silently altering the file content. In particular, we consider the scenario in which a malicious user u_k aims at fraudulently modifying a certain file $F = (V_0, \dots, V_l)$ in one of the following possible ways:

1. pushing some changes into the file without creating a new version, i.e., changing the content of an existing version;
2. changing the author of an existing intermediary version, e.g., so that a version initially created by a revoked user can be afterwards attributed to a different author (e.g., u_k);
3. deleting an existing intermediary version or creating a new version between two existing consecutive versions;

Let file F be composed of versions $V_0, \dots, V_l$, and consider an insider attacker $u_k \in G, u_k \neq s_l$. We show how TVFS prevents u_k to successfully carry out the attacks listed above. We recall that the Content Extraction Signature $\mathcal{C}$ implemented by TVFS satisfies the CES-Unforgeability property (see Section 2.1), as the standard signature scheme $\mathcal{S}$ it is based upon, statisfies the standard unforgeability property.

1. **An adversary cannot push changes into an existing version.** Suppose that u_k aims at tampering with the content of the record representing any existing intermediary version V_i, where $0 \leq i < l$. In particular, the adversary may attempt to change either the content of any block b_j, (including h_i), delete any block b_j or add a new block $\bar{b}$ to V_i.
   We first argue that TVFS does not allow u_k to modify any block $b_j \in V_i$, $\forall 0 \leq i < l$. To this end, recall that V_i is represented by the undo record *undo(V_{i+1})* and that each block $b_j \in$ *undo(V_{i+1})* is protected by signatures $\sigma^{(i)}_{full}$, computed by the signer s_i and stored in h_i and $\sigma^{(i)}_{undo}$ extracted by signer s_{i+1} and stored in h_{i+1}. Moreover, the integrity of block h_{i+1} in turn, is assured by the signature $\sigma^{(i+1)}_{full}$ computed by s_{i+1}.
   We firstly consider that $s_i \neq s_{i+1}$. If $u_k \neq s_i$ and $u_k \neq s_{i+1}$, in order to modify any block $b_j \in V_i$, u_k should be able to forge $\sigma^{(i)}_{full}$ and $\sigma^{(i+1)}_{full}$, but this is not possible due to the standard unforgeability of $\mathcal{S}$. If $u_k = s_i$, she can change the content of b_j and recompute $\sigma^{(i)}_{full}$ and $\sigma^{(i)}_{undo}$ but cannot replace the latter in h_{i+1} since she cannot forge $\sigma^{(i+1)}_{full}$. Conversely, if $u_k = s_{i+1}$, upon changing b_j, the adversary can neither forge $\sigma^{(i)}_{full}$ nor $\sigma^{(i)}_{undo}$.
   We stress that the adversary can neither add a new block $\bar{b}$, nor delete any existing block b_j from version V_i, since both these actions entail the re-computation of the involved signatures as well as the modification of metadata contained in the blocks h_i and h_{i+1}.
   Consider the case in which $u_k = s_i = s_{i+1}$ or, more generally, consider that u_k is managing to make fraudulent changes into a sequence of d intermediary versions $V_i, V_{i+1}, \dots, V_{i+d}$ which she signed herself. According to the assumptions we made above, we have $0 \leq i < i + d < j = (i + d + 1) \leq l$ and $u_k \neq s_j$ so that u_k may produce a sequence of tampered versions $V'_i, \dots, V'_{j-1}$ for F. However, the header block h_j contains the extracted signatures $\sigma^{(j)}_{undo}$ and $\sigma^{(j)}_{old}$, both extracted from the full signature $\sigma^{(j-1)}_{full}$ of the original version V_{j-1}. Let $\bar{\sigma}$ be the full signature of the tampered version V'_{j-1}, in order to push her changes to the file, u_k should be able to extract $\sigma^{(j)}_{undo}$ and $\sigma^{(j)}_{old}$ from $\bar{\sigma}$ but, due to the CES-unforgeability of $\mathcal{C}$, this is not possible, unless $\bar{\sigma} = \sigma^{(j-1)}_{full}$, that is $V_{j-1} = V'_{j-1}$.
2. **An adversary cannot change the author of an existing version.** Indeed, in order to replace the

signer s_i in V_i, u_k should be able to silently push several changes to metadata contained in h_i and h_{i+1}.

3. **An adversary cannot delete or create intermediary versions.** Indeed, deleting an intermediary version V_i, requires u_k to make several changes to the header block of version V_{i+1}. In particular, note that the header block h_i of the vanishing version V_i is contained in *undo(*V_{i+1}*)* whose extracted signature is stored in h_{i+1}. Consider that *undo(*V_{i+1}*)* and *undo(*V_i*)* necessarily differ as $h_i \neq h_{i-1}$.
Analogously, inserting a new version V_i' between V_i and V_{i+1} would require u_k to either change extracted signatures in h_{i+1} or make V_i' include all blocks in *undo(*V_{i+1}*)*, but this is not possible since h_i belongs to *undo(*V_{i+1}*)* but not to V_i'.

7. Prototype implementation

In this section, we briefly present a proof-of-concept prototype implementation of TVFS and discuss main issues raised by this task, as well as the achieved performance.

The major properties that make Mobile Storage Devices so widely appreciated are probably their extreme flexibility and portability, that allow users to transport and exchange files without any particular restriction or slowness. Therefore, in designing TVFS our main aim was seconding these wishes, making possible to share multiple-version files almost like plain files, among the widest possible set of platforms.

To this end, we chose to implement TVFS as a user-level file system layer leveraging on the FUSE framework [19], so that its additional functionalities, could be transparently mapped to the storage device's chosen file system, achieving substantial independency from the user's preferences. Moreover, we chose to implement TVFS using Python [20], a well known and consolidated high-level programming language that provides a stable interface with the FUSE framework. We remark that implementing the TVFS prototype using these two tools enables good OS portability, as FUSE is currently under development for several mainstream operating systems, including Linux [19], Mac OS X [21], and Windows [22, 23].

Our early experiments aimed at measuring the performance of a real-world implementation of the Content Extraction Signature scheme. Although theoretically, CES should perform better than schemes based on batch signature, we wondered if, once implemented, it would also offer high performance in practice. As a first step, we just implemented the scheme as is proposed in [11]. The implementation of the Content Extraction Access Structure was the main matter of concern. This structure is used, during extracted signature verifications, to check if the extracted subdocument is composed of a legitimate subset of the full document's blocks, in other words the verifier check whether the set of the subdocument's indices belongs to the CEAS. Therefore, the CEAS is described as an arbitrary subset of the powerset of the document's block indices. This object should be handled carefully, for two reasons: (1) it is attached to the signature, hence, its representation strategy affects the overall file system performance in terms of storage space, (2) membership queries to rather large CEASes may produce unacceptable overhead. In our prototype we chose to replace the CEAS with a list of "mandatory blocks" that contains the blocks of the original document that must be present in every extracted subdocument. Though this model is less general than the original (that enables the signer to impose more sophisticated extraction policies), it is sufficient for the sake of our experiments and performs remarkably better. Figure 4 summarizes results of our experiments with a full-python implementation of TVFS. In particular, Figure 4(a) shows the CES signature size, whereas Figures 4(b), 4(c) and 4(d) give a view over the scheme's signature, extraction and verification times. Graphs in Figures 4(e) and 4(f) show respectively the time needed to check in a new version (*e.g.*, as it happens when the user writes some changes to the file), and to check-out the latest version from a two-versions file, that is the time spent in each iteration of the VALIDATE() algorithm.

The experiments have been conducted on an ordinary PC equipped with a processor AMD Athlon$^{\mathrm{TM}}$II X2, 4 Gigabytes of RAM and two hard disks Serial ATA.

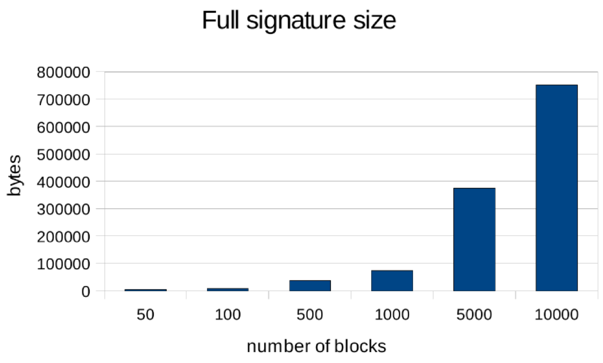
We find that achieved experimental results are quite encouraging and show that our approach is practical and is suitable to its target scenario. Performances are rather good, considering that, usually, MSDs perform worse than traditional hard disk so that, the overhead due to check-in and check-out operations is partially absorbed by the device's latency.

The main drawback of our approach is that using a bodied interpreted language such as Python significantly lowers performances, especially in term of I/O throughput. On the other hand, we point out that Python offers a good interface to native languages like C and C++ (as its variant jython does to the java world), so that it could be possible to implement some time-critical tasks with platform-dependent components, though embedded in the overall high-level infrastructure. Some experiments we did in this direction produced promising results.

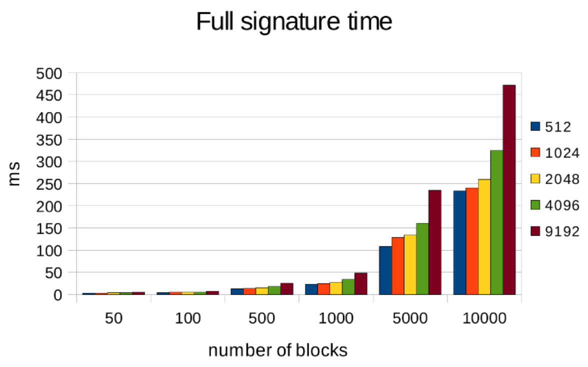
8. Related Work

Versioning file systems have a more than thirty years long history. Among the earliest solutions, we mention the file system of the long-lived VMS [24] operating system, which since the eigthies allows users to handle multiple versions of their files, the Cedar [25] and the Elefant [26] file systems.

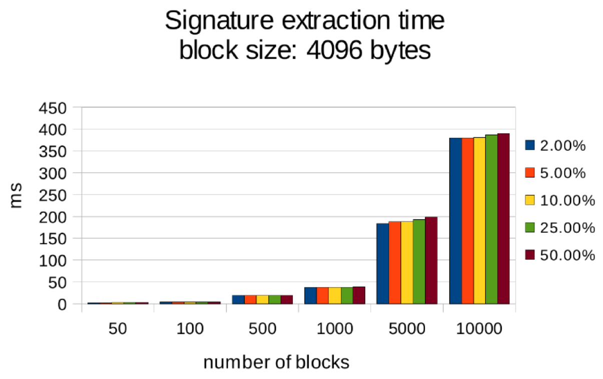
Versioning file systems usually fall in two categories: *continuous versioning* and *snapshotting* file system, according to how and when new versions are created. Continuous versioning file systems such as CVFS [29], VesionFS [27] and Wayback [30], create a new version of a



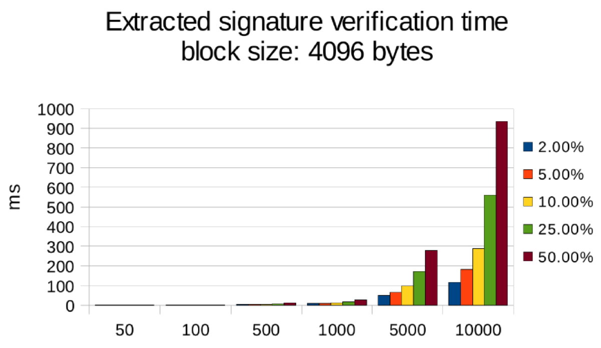
(a) Size of the signature of full documents composed of 50, . . . ,5000 blocks. The Signature size is independent from the block size.



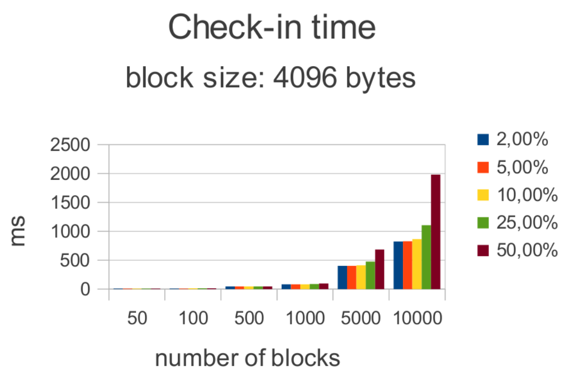
(b) Time employed to compute the signature of full documents composed of 50, . . . , 5000 blocks. Time performance was measured on files with blocks of different sizes (512, . . . , 4096 bytes).



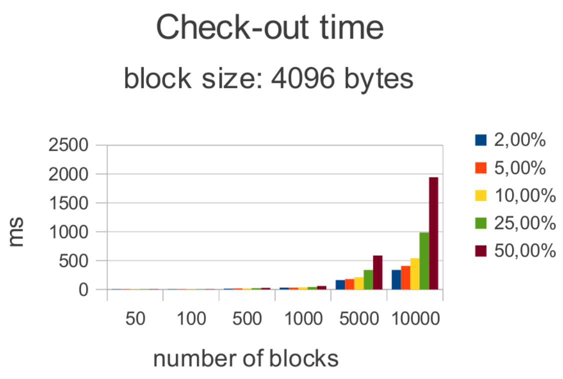
(c) Time employed to extract the signature for subdocument composed of different subsets of blocks from an original document whose block size is 4096 bytes.



(d) Time employed to verify the extracted signature for subdocument of different sizes.



(e) Time employed to check in a new version



(f) Time employed to check out the latest version from a two-versions file

Figure 4: Content Extraction Signature: performance summary.

Name	Main features	Architecture	Requirements	Trust model	Applicable to MSDs	Type	Key Mgmt
TVFS	File encryption; trustworthy data retention, verifiable history in a seamless interval of time	The repository lies on an untrusted MSD		Trusted client OS, untrusted repository	Trusted client OS, untrusted repository	Cont.	Generic PKI plus encryption key distribution service
VDisk[5]	File-grained versioning; reliable log mgmt.	The client is a compartment running a legacy OS. Versioning is implemented as a layer below the Xen virtual driver interface; user-level applications accomplish versioning and logging tasks.	XEN based virtualized infrastructure	The hardware is assumed physically secure. The virtualization layer is trusted. The client OS and the user-level components are untrusted	Not applicable	Cont.	n/a
Ext3Cow with audit[2]	Extends the Ext3 file system witha a mechanism which features verifiable digital audit trials.	When a new version is committed, a MAC for the change is generated and published to a trusted third party which maintains the file history information.	Remote audit server managed by a trusted third party	Auditor relies on the audit server. The file storage is untrusted	Not applicable, unless a mechanism to distinguish among different clones of the storage devices is provided.	Snap.	Users are provided of a public/private key pari by an external PKI. File encryption keys are stored in "lockboxes" encrypted with the user's private key.
Ext3Cow with secure del.[3]	Extends the Ext3 file system with a mechanism for secure deletion of individual file snapshots.	Features block-grained data encryption with authenticated encryption. A portion of the resulting cyphertext(stub) is stored as metadata and is the only part which is processed when a secure deletion occurs.		The client OS is trusted; It is assumed that the HD is physically secure	Not applicable. An MSD can be arbitrarily removed and cloned.	Snap.	As in [2]
SVSDS[4]	Selective file-grained versioning; Unauthorized changes reversibility; enhanced file access control.	Leverages on a dedicated storage device whose firmware implements the version control system and a trusted administrative interface.	Dedicated storage device.	The storage device is trusted, the client OS is untrusted	Not applicable on untrusted off-the-shelf MSD.	Snap.	N/a
VersionFS[27] with Truecrypt[28]	Plain versioning file system layer mounted on top of an encrypted MSD.	The repository lies on an MSD that is plugged to a personal workstation.		Trusted client OS	Trusted client OS	Cont.	Relies on an external key management infrastructure.

Figure 5: Related work, main features summary.

	History reliability	Data integrity	Data secrecy	Data retrievability	Change immutability	Change undeniability	Version merging/pruning	Content revocation (per user)	Per version encryption	Per file encryption	Per block encryption	Per device encryption	Trusted client OS	Trusted third party	Trusted storage device	Trusted remote storage server	Device FS independence	Client OS Independence	Storage device independence
TVFS	X	X	X	X	X	X		X	X	X		X	X				X	X	X
VDisk[5]	X			X	X		X								X			X	
Ext3Cow+audit[2]	X	X		X	X	X								X					X
Ext3Cow+secure del.[3]		X	X				X		X		X		X						X
SVSDS[4]	X			X	X	X									X			X	
VersionFS[27]+Truecrypt[28]			X				X					X	X						

Figure 6: Features comparison

file, every single time it is modified, so that the set of all version of a file seamlessly represents its history. So called snapshotting file systems like Ext3Cow [31], retain multiple versions of the whole files system (snapshots), created automatically at regular intervals of time. However, snapshotting file systems cannot keep track of changes made between two snapshots.

Pursuing the regulatory compliance has been one of the main guidelines to several secure versioning file system projects. Peterson *et al.* proposed two cryptographic extensions of Ext3Cow which feature secure deletion [3] and verifiable auditing [2]. More in general, the main goal of a secure versioning file system is to guarantee file version integrity, retrievability and historical log reliability. To do so, SVSDS [4] and VDisk [5], push file block-grained versioning into the boundaries of a Trusted Computing Base respectively identified with the disk drive firmware or a secure Virtual Machine Monitor which embodies a virtual storage device.

Designing a versioning file system is still a challenging task, as it is essentially a matter of achieving an acceptable tradeoff among I/O throughput, storage space employment and, having deal with sensitive data retaining, security requirements. The latter aspect, in particular, highly impacts on the overall architecture. Moving the versioning functionality out of the Operating System as in SVSDS and VDisk allows to keep the data repository rather far from the hands of potential intruders, as well as implementing such functionalities at such a low level may significantly improve performance. However, this approach does not fit those application scenarios in which the data repository lies on mobile or untrusted storage devices, as we focus on in this paper. Analogously, leveraging on a trusted third party, as happens in [3] and [2] for versioning operations, poses several problems with respect to the traditional way MSD are used. For example, allowing users to freely make multiple copies of a repository could affect the capacity to log the actions done to any file it contains, coherently.

A very basic approach to enhance the security of a plain versioning file system, within our target scenario, is storing its repository on an encrypted volume [32, 28]. However, this solution lacks of flexibility as it does not allow to implement fine-grained access control policy and poses some non trivial issues to handle users/keys revocation. Figure 5 summarizes main characteristics of some of related works cited above. A feature-by-feature comparison with TVFS is shown in Figure 6.

As far as we know, TVFS is one of the earliest versioning file systems, designed to explicitly cope with security issues raised by managing data stored on untrusted storage devices as MSDs.

9. Conclusion

In this paper we present the design and the implementation of a novel Trusted Versioning File System (TVFS) that achieves confidentiality and integrity of stored files, as well as the authenticity of their change history in a certain seamless interval of time. Unlike the majority of existing secure versioning file systems, TVFS mainly targets those application scenarios in which the data repository resides on untrusted Mobile Storage Devices, such as forthcoming e-health applications that increasingly delegate patient's data storage to personal held devices.

TVFS security leverages on the Content Extraction Signature scheme, to implement its data repository. In this paper we describe the overall architecture TVFS aims to fit, the data repository's structure, the versioning process and the file signature scheme. Eventually, we analyze its security achievements.

We present a proof-of-concept prototype which implements a user-level file system layer that can work independently from the underlying storage technology and file system type. It is written in a high-level programming language, achieving high flexibility and portability. Early performance measurements are also presented and discussed. We belief that our results promise remarkable improvements.

Although we explicitly address the problem of enforcing access control over USB drives and similar devices, our solution could be extended to distributed architectures. For instance, in a cloud computing infrastructure the file repository can be hosted by a third party service provider and can be accessed through the network.

References

[1] Berliner, Polk, Concurrent versions system (cvs), http://www.cvshome.org/ (2001).

[2] Z. Peterson, R. Burns, G. Ateniese, S. Bono, Design and implementation of verifiable audit trails for a versioning file system, in: Proceedings of the 5th USENIX conference on File and Storage Technologies table of contents, USENIX Association Berkeley, CA, USA, 2007, pp. 20–20.

[3] Z. Peterson, R. Burns, J. Herring, A. Stubblefield, A. Rubin, Secure deletion for a versioning file system, in: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies-Volume 4, USENIX Association Berkeley, CA, USA, 2005, pp. 11–11.

[4] S. Sundararaman, G. Sivathanu, E. Zadok, Selective versioning in a secure disk system, in: Proceedings of the 17th USENIX Security Symposium, USENIX Association, 2008, pp. 259–274.

[5] J. Wires, M. Feeley, Secure file system versioning at the block level, in: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, ACM New York, NY, USA, 2007, pp. 203–215.

[6] H. Löhr, A.-R. Sadeghi, M. Winandy, Securing the e-health cloud, in: Proceedings of the 1st ACM International Health Informatics Symposium (IHI 2010), ACM, 2010, pp. 220–229. doi:http://doi.acm.org/10.1145/1882992.1883024.
URL http://doi.acm.org/10.1145/1882992.1883024

[7] M. Winandy, A note on the security in the card management system of the german e-health card, in: Proceedings of the 3rd International ICST Conference on Electronic Healthcare for the 21st Century (eHealth 2010), Springer, 2010, pp. 196–203.

[8] I. M. of Environment, Italian waste tracking system (sistri), http://www.sistri.it (2009-2011).

[9] R. Steinfeld, L. Bull, Y. Zheng, Content extraction signatures, in: Information Security and Cryptology - ICISC 2001, 4th International Conference Seoul, Korea, December 6-7, 2001, Proceedings, Vol. 2288 of Lecture Notes in Computer Science, Springer, 2002, pp. 285–304.

[10] R. Steinfeld, L. Bull, Y. Zheng, Content extraction signatures (full version), http://www.sis.uncc.edu/~yzheng/publications/files/CES_full-2003.pdf (2003).

[11] L. Bull, D. McG. Squire, Y. Zheng, A Hierarchical Extraction Policy for content extraction signatures, International Journal on Digital Libraries 4 (3) (2004) 208–222.

[12] S. Goldwasser, S. Micali, R. Rivest, A digital signature scheme secure against adaptive chosen-message attacks, SIAM J. Comput. 17 (2) (1988) 281–308.

[13] W. Tichy, Design, implementation, and evaluation of a Revision Control System, in: Proceedings of the 6th international conference on Software engineering, IEEE Computer Society Press Los Alamitos, CA, USA, 1982, pp. 58–67.

[14] M. Backes, C. Cachin, A. Oprea, Lazy revocation in cryptographic file systems, in: 3rd International IEEE Security in Storage Workshop (SISW 2005), December 13, 2005, San Francisco, California, USA, 2005, pp. 1–11.

[15] D. Naor, A. Shenhav, A. Wool, Toward securing untrusted storage without public-key operations, in: Proceedings of the 2005 ACM Workshop On Storage Security And Survivability, StorageSS 2005, Fairfax, VA, USA, November 11, 2005, ACM, 2005, pp. 51–56.

[16] M. Backes, C. Cachin, A. Oprea, Secure key-updating for lazy revocation, in: Computer Security - ESORICS 2006, 11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18-20, 2006, Proceedings, Vol. 4189 of Lecture Notes in Computer Science, Springer, 2006, pp. 327–346.

[17] L. Catuogno, M. Manulis, H. Löhr, A.-R. Sadeghi, M. Winandy, Transparent mobile storage protection in trusted virtual domains, in: 23rd Large Installation System Administration Conference (LISA'09), USENIX Association, 2009.

[18] G. Singaraju, B. H. Kang, Concord: A secure mobile data authorization framework for regulatory compliance, in: Proceedings of the 22nd Large Installation System Administration Conference, LISA 2008, November 9-14, 2008, San Diego, CA, USA, USENIX Association, 2008, pp. 91–102.

[19] M. Szeredi, File system in user space, http://sourceforge.net/projects/fuse.

[20] G. V. Rossum, The Python Language Reference, Python Software Foundation, 1990-2011, http://www.python.org.

[21] B. Fleischer, E. Larsson, FUSE for OS X, https://osxfuse.github.com.

[22] Dokan Project, Dokan – User mode filesystem for windows, http://dokan-dev.net/en.

[23] EldoS Corporation, Callback file system, http://www.eldos.com/cbfs.

[24] K. McCoy, VMS file system internals, Digital Press, 1990.

[25] D. Gifford, R. Needham, M. Schroeder, The cedar file system, Communications of the ACM 31 (3) (1988) 288–298.

[26] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, J. Ofir, Deciding when to forget in the elephant file system, ACM SIGOPS Operating Systems Review 33 (5) (1999) 110–123.

[27] K. Muniswamy-Reddy, C. P. Wright, A. Himmer, E. Zadok, A Versatile and User-Oriented Versioning File System, in: Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004), USENIX Association, San Francisco, CA, 2004, pp. 115–128.

[28] TrueCrypt Foundation, Truecrypt - free open-source on-the-fly encryption, http://www.truecrypt.org/ (2004).

[29] C. Soules, G. Goodson, J. Strunk, G. Ganger, Metadata Efficiency in Versioning File Systems, in: Proceedings of the 2nd USENIX Conference on File and Storage Technologies, USENIX Association Berkeley, CA, USA, 2003, pp. 43–58.

[30] B. Cornell, P. Dinda, F. Bustamante, Wayback: A user-level versioning file system for linux, in: Proceedings of Usenix Annual Technical Conference, FREENIX Track, 2004, pp. 19–28.

[31] Z. Peterson, R. Burns, Ext3cow: a time-shifting file system for regulatory compliance, ACM Transactions on Storage (TOS) 1 (2) (2005) 190–212.

[32] Microsfot Corp., Bitlocker drive encryption, http://technet.microsoft.com/en-us/windows/aa905065.aspx (2006).