sec-cs: Getting the Most out of Untrusted Cloud Storage

Dominik Leibenger and Christoph Sorge
CISPA, Saarland University

Abstract

We present `sec-cs`, a hash-table-like data structure for file contents on untrusted storage that is both secure and storage-efficient. We achieve authenticity and confidentiality with zero storage overhead using deterministic authenticated encryption. State-of-the-art data deduplication approaches prevent redundant storage of shared parts of different contents irrespective of whether relationships between contents are known a priori or not.

Instead of just adapting existing approaches, we introduce novel (multi-level) chunking strategies, ML-SC and ML-CDC, which are significantly more storage-efficient than existing approaches in presence of high redundancy.

We prove `sec-cs`'s security, publish a ready-to-use implementation, and present results of an extensive analytical and empirical evaluation that show its suitability for, e.g., future backup systems that should preserve *many* versions of files on little available cloud storage.

1 Introduction

Cloud storage solutions have become increasingly popular among customers. They usually provide a limited amount of storage space that is accessed over the internet and used for synchronization of personal data between devices or for backup purposes, e.g., by using *rsync* [39] to synchronize a user's important data to the cloud storage in frequent intervals. Ideally, cheap creation of snapshots should be supported, i.e., the user should be able to preserve lots of consistent copies of specific states of her backed up data without being charged for redundant or duplicate data. Such snapshots/versioning features are provided by many cloud storage providers, e.g., Dropbox [7].

Unfortunately, *security guarantees* of today's popular providers are insufficient: Malicious providers could read and modify data unnoticed by their users. The above-described scenario, thus, requires application of cryptographic measures on the client side as to ensure confidentiality and authenticity of outsourced data. Secure encryption using tools like GnuPG [13], however, hides any information about file contents—including differences across versions—from the provider, thus preventing any savings from its snapshots feature. Only few systems try to combine security and storage efficiency; neither is both secure and able to provide efficiency comparable to unencrypted cloud storage to the best of our knowledge. Consequentially, users have to decide between cheap & comfortable and expensive & secure solutions today.

As we consider both aspects equally important, our goal is to advance development of practical solutions (e.g., backup systems) for cloud storage with strong security and better efficiency guarantees. To this end, we present a novel data structure for file contents on untrusted cloud storage, `sec-cs`, with the following contributions:

- We design and integrate a novel chunking-based data deduplication concept, ML-*, that outperforms existing approaches w.r.t. storage efficiency when storing contents with high redundancy.
- We achieve strong confidentiality and authenticity guarantees of stored data with zero storage overhead.
- We further publish a ready-to-use implementation and evaluate its efficiency analytically and empirically, proving superiority to other approaches.

This paper is structured as follows: We give background information and present related work in Sec. 2. ML-* is introduced in Sec. 3 and detailed as part of `sec-cs`, which is described in Sec. 4 and proven secure in Sec. 5. Our implementation is discussed in Sec. 6 and an evaluation is presented in Sec. 7. Sec. 8 concludes the paper.

2 Background and Related Work

As both *data deduplication* (i.e., elimination of redundancy across stored data), and security are essential goals of `sec-cs`, different kinds of existing work are related.

2.1 Data Deduplication

Existing deduplication systems apply some deterministic chunking scheme $\mathcal{C}$ to a content to split it into non-overlapping chunks, and avoid storage of resulting chunks that have already been stored before, usually by maintaining an index of cryptographic hash values of chunks. An overview and a classification of common approaches and their efficiency is provided by Meister and Brinkmann [25]: *Whole-file chunking (WFC)* yields a single chunk and is thus able to detect identical file contents. *Static chunking (SC)* splits a content into chunks of fixed size that are individually deduplicated, so partially overlapping contents can be deduplicated as well. It is used, e.g., in Venti. [30] *Content-defined chunking (CDC)*, in contrast, can even tolerate shifted contents. It determines chunk boundaries by moving a sliding window of some fixed size W over the content and creating chunk boundaries when a window content meets a specific criterion, typically a hash value being in a specific range. This yields chunks of some *expected* length—the *target chunk length*—under the assumption that different positions have different window contents and hash values are uniformly distributed. To deal with non-uniformly distributed contents, a minimum and maximum chunk size can be set. The scheme was introduced by Muthitacharoen et al. [28] for the low-bandwidth network file system and is usually implemented using a rolling hash, e.g., Rabin fingerprints [31]. Alternatives to the basic sliding window approach usually used for CDC that might be worth consideration for future enhancements of `sec-cs` are presented by Eshghi and Tang [10]. According to [25], SC yields better deduplication efficiency than WFC and CDC is more efficient than SC for real-life data. Few systems like ADMAD [23] are able to achieve even better efficiency by employing *application-specific chunking (ASC)*. ASC, however, requires additional knowledge about the respective file format of a content.

Instead of simply avoiding multiple storage of identical chunks, some systems employ *delta encoding*: When a highly similar chunk—a *base chunk*—is known, a new chunk is represented as a reference to the base chunk and a *difference*, i.e., a sequence of actions that define how to create it from the base chunk. In combination with WFC, this scheme is, e.g., used in version control systems (VCS) like Subversion (SVN) [38]. SVN's *FSFS* backend stores the first revision of a file content in its entirety and all subsequent revisions as differences to previous revisions. [4]

A comparison of advantages of the different schemes is shown in Tab. 1: Delta-based approaches are clearly able to yield lowest storage costs for changed contents in principle, but they have important limitations in practice: First, their efficiency depends on a priori knowledge of relations between chunks—a problem tackled by, e.g., the DERD framework [6]. Second, they substantially increase retrieval costs as reconstruction of delta-encoded chunks requires retrieval of corresponding base chunks of which only parts are actually required. Chunking-based schemes achieve savings when storing changed contents (data) irrespective of knowledge about relations (their best values are highlighted in green in the table). The most efficient

Storage costs for...	Delta	WFC	SC	CDC	ASC	ML-SC	ML-CDC
single file (data)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
single file (metadata)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
single file (Σ)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
change w/o shift (D)	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
change w/o shift (M)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
change w/o shift (Σ)	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
change w/ shift (D)	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
change w/ shift (M)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
change w/ shift (Σ)	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
Efficiency is independent from file content's...							
size	✓	✓	✗	✗	✗	✓	✓
uniform distribution	✓	✓	✓	✗	✗	✓	✗
precise format	✓	✓	✓	✓	✗	✓	✓
relation to others	✗	✓	✓	✓	✓	✓	✓

Table 1: Efficiency of dedup. concepts (n: content length)

strategies CDC / ASC yield their savings depending on the distribution of contents / specific file formats.

In addition to the costs for the actual content *data*, every data deduplication mechanism incurs storage costs for *metadata*: In case of WFC, this is only a small constant per content corresponding to its cryptographic hash value. Chunking-based schemes do not only incur this overhead for every chunk, but they also require additional storage space to store of which chunks a specific content consists—typically a list of chunk identifiers, e.g., cryptographic hash values. Although the respective constants are small for large chunk sizes, metadata storage costs for contents deduplicated via SC, CDC and ASC are linear in their lengths no matter how high their redundancy is. These scheme's chunk size parameters, thus, considerably impact their storage efficiency, as also noticed by Eshghi and Tang [10]: If set too high, deduplication performance is decreased, as only completely identical chunks allow space savings. If set too small, storage of contents with high redundancy cause considerable overhead due to the sheer amount of chunk references that have to be stored.

This problem is solved by our proposals ML-SC / ML-CDC: Due to a specific, multi-level application of SC / CDC, we achieve logarithmic metadata costs, allowing high storage efficiency even for small chunk sizes, independent from content sizes. To the best of our knowledge, our strategy is unique. Teodosiu et al. [37] apply CDC recursively to enable efficient replication of files over a network (assuming the receiver has a similar file to the one being transferred), but they do not target storage systems and fail to achieve logarithmic costs due to a fixed recursion depth. Further, their approach is different to ours: Instead of breaking large chunks recursively into smaller ones, they generate the smallest chunks first and use CDC recursively to break lists of chunk references into smaller parts—requiring multiple passes. Yasa and Nagesh [1] employ hierarchical chunking starting with CDC at the highest level as we do, but they use only two levels (SC at second level), so storage overhead is still linear.

2.2 Security

Lots of works exist in the related fields of cloud security and cryptographic file systems, but only few focus on authenticity and storage efficiency. Popular cloud storage security solutions typically deal with confidentiality only. BoxCryptor [3], e.g., is based on and uses a similar concept to EncFS [9]: It encrypts file contents symmetrically, but does not provide authenticity. While not preventing storage-efficient snapshots of unchanged files, costs for changed files are high due to entirely different ciphertexts.

SiRiUS [14], Plutus [19] and Tahoe-LAFS [42] are examples of file systems with authenticity guarantees: They apply SC to contents and compute a Merkle tree [26] over the chunks to allow authenticity verification even for parts of contents, but they do not support data deduplication: Tahoe-LAFS creates entirely different ciphertexts for similar file contents, the other systems even for identical ones.

To allow efficient usage of cloud storage for, e.g., backups, more specialized tools are required. Common backup tools like duplicity [8] rely on *incremental* backups, i.e., they store differences to previous backups. This can be used in combination with GnuPG to preserve snapshots in a storage-efficient and secure manner, but causes communication overhead when specific versions are read. VCS could be used for delta-based backups to a limited extent, but they are typically inefficient w.r.t. large files and have limited security properties: Git [12] only ensures integrity/authenticity of the version history by integrating signatures. SVN does not, but an extension [22] adds storage-efficient file-level encryption. Cumulus [41] is a backup system that supports large files and allows direct access to arbitrary snapshots, but it is less storage-efficient as it only deduplicates identical data between different versions of individual files. Farsite [5], in contrast, is a distributed file system targeting on chunking-based backups that deduplicates different identical files despite secure encryption. It cannot save space for snapshots of different versions of a file, though, as it relies on WFC. Storer et al. [35] extend Farsite's concept to CDC, but they do not provide any explicit authenticity guarantees.

Many more works exist in the field of cloud storage security. Most of them, however, have a different focus and are orthogonal to our work. Athos [15], e.g., is a solution for outsourcing file systems that achieves integrity in a way that file system operations are possible with logarithmic communication costs. The solution is orthogonal to our work in the sense that this requirement is w.r.t. the total number of files/directories in the file system and not w.r.t. the size of single file contents. A similar goal is pursued by Heitzmann et al. [18]. Both works are based on authenticated skip lists, an authenticated data structure (ADS) initially proposed by Goodrich et al. [16], while our work is based on another ADS—the Merkle tree [26].

ADS, in general, is an umbrella term for data structures involving three parties (a trusted *source* who publishes data, an untrusted *responder* that stores structured data, and a *user* that requests data) that enable authenticated, efficient queries to the data. [36] In this sense, `sec-cs` can be considered an ADS with additional data deduplication and confidentiality properties: The cloud storage backend can be considered the *responder* and the user/client plays the roles of *source* and *user*. An overview of existing ADS and methods for constructing ADS in general are provided by Martel et al. [24] and Miller et al. [27].

3 ML-* – Multi-Level Chunking

Our first contribution are the chunking strategies ML-SC and ML-CDC which improve on the state of the art in terms of storage efficiency in presence of high redundancy (see Tab. 1). The basic idea is simple: As the linear overhead for storing contents with a traditional strategy $\mathcal{C}$ is caused by the need of storing references to each constant-size part of each content, we want to also deduplicate these references. This can be achieved by representing the results of $\mathcal{C}$ on a content m as a *chunk tree* t, whose

- *leaf nodes* represent the chunks output by $\mathcal{C}$, and
- *inner nodes* aggregate chunks, representing the concatenation of chunks represented by their children.

In addition to *leaf chunks* (chunks represented by leaf nodes), we thus create "larger" *superchunks* (chunks represented by inner nodes), which we persist as well and which can be referenced directly when new contents are stored. Consequentially, each content is represented by one persisted *root chunk* (the chunk represented by the tree's root node, which might be a leaf or superchunk).

Persisting a superchunk requires storing references to its children. To ease notation, we refer to storage costs of a chunk representation as its *size* and to the length of its represented content as its *length*. For leaf chunks we assume size is equal to length. To enable high storage efficiency, we require *sublinear storage overhead* for storing a content m' having large overlaps with an existing content m. For this, we generate their chunk trees t, t' so that the following *requirements* are met:

R1: the (expected) size of each chunk is constant,

R2: identical parts of m and m' share not only leaf, but also superchunks (i.e., t and t' share subtrees), and

R3: the heights of t and t' are chosen logarithmically in the lengths of m and m', respectively.

Thus, if m' differs from m in only one byte, t and t' shall be equal except for one chunk at each level, so that their difference consists of $\mathcal{O}(\log |m|)$ constant-size chunks.

Different chunking strategies allow to achieve this. In the simplest case, we could aggregate fixed numbers of consecutive chunks output by SC to superchunks, and continue aggregating fixed numbers of subsequent superchunks until only a single superchunk—the root chunk—remains. This approach, however, would eradicate advantages of chunking schemes that go beyond those of SC. While leaf chunks output by CDC, for example, are robust against shifting, this property would not be true for superchunks. To account for that, we define our multi-level chunking scheme in a more general way that preserves the properties of its underlying chunking algorithm $\mathcal{C}$. The only requirements we state is that $\mathcal{C}$ has to be deterministic and that it has a parameter S that allows to specify the target (or expected) length of its generated chunks. Now let $R < S$ be the size required for representing a single *chunk reference*, i.e., anything that allows retrieval of the corresponding chunk (R is constant as chunks will be referenced by hash values). We define ML-$\mathcal{C}$ as follows:

- On input a content m with length $n = |m|$, choose the height h of the to-be-built chunk tree t as

$$h = \min_{h' \in \mathbb{N}} \left\{ h' \;\middle|\; n \leq \frac{S^{h'+1}}{R^{h'}} \right\} = \left\lceil \frac{\log\left(\frac{n}{S}\right)}{\log\left(\frac{S}{R}\right)} \right\rceil, S > R \quad (1)$$

  where $h = 0$ describes a single-node tree.
- Create root node of t that should represent m.
- Iterate over the nodes of the tree in a breadth-first search manner. For each node with height $h' > 0$ (beginning with the root node having height $h' = h$),
  - determine content $\tilde{m}$ that the node represents,
  - apply the chunking strategy $\mathcal{C}$ on $\tilde{m}$ with target chunk length $\frac{S^{h'}}{R^{h'-1}}$, and
  - add child node for each chunk $\tilde{m}'$ output by $\mathcal{C}$.

This way, a content smaller than or equal to the target chunk size S results in a single leaf node, and for $n \geq S$, all leaf chunks have target chunk size S. As superchunks at height h' have expected length $\frac{S^{h'+1}}{R^{h'}}$ and are chunked with target chunk length $\frac{S^{h'}}{R^{h'-1}}$, they are expected to have $\frac{S}{R}$ children. As child references have size R, the expected size of superchunks is S as well, fulfilling Req. R1. Req. R3 is met by the choice of h and Req. R2 is expected to be achieved due to straightforward application of $\mathcal{C}$ at each level. The latter is concretized in Sec. 4.4.1 and discussed and evaluated in detail in Sec. 7.

4 sec-cs – The Secure Content Store

For a detailed analysis, we embed ML-* in a generic data structure that is described in this section. `sec-cs` acts like a normal hash table that assigns a deterministically computed hash value to each inserted content, but comes with a combination of properties different from prior work: It employs *multi-level chunking* to significantly reduce storage overhead for large overlapping contents and it guarantees *authenticity* and *confidentiality*.

Note that `sec-cs` is limited to immutable contents, i.e., it does not support deletion of contents. We present this variant as it is sufficient for the evaluation of ML-*, but we emphasize that it is easy to extend it to a mutable variant, either by allowing deletion of root nodes (requiring a garbage collection for non-referenced chunks), or using reference counters for chunks (at the cost of some slight storage overhead). In fact, our implementation (see Sec. 6) supports the latter.

4.1 Prerequisites

`sec-cs` requires a backend to persist data. Low-level storage management is out of scope of this paper, though. Instead, we assume the existence of a backend providing the following *key-value store (KVS)* interface:

- PUT(k, v)—persist value v under key k
- GET(k)—return v or $\perp$ if key k does not exist

Note that the KVS interface can be easily mapped to any commonly-used storage backend: Key-value databases and many cloud (object) storage providers can be accessed by this interface and a mapping to a file/directory structure in a file system could be done in a straightforward manner. The major requirement is that the backend can deal efficiently with many key/value pairs.

4.2 Threat Model

The goal of `sec-cs` is to allow efficient and secure usage of existing cloud storage for storing file contents, especially in presence of many (similar) versions of contents and an untrusted cloud storage provider. Towards this goal, our model includes two parties, a user (client) and a backend (server). The user is assumed to be completely trustworthy: She instantiates the data structure and locally executes its operations in order to change its state. Any operation invocation done by the user is considered legitimate. The user is required to locally store and keep secret a fixed number of constant-size cryptographic keys. The backend does not need to be trustworthy at all. It might read any stored data and also write, overwrite or delete any data as to perform malicious modifications (e.g., changes to file contents) that remain undetected by the user. The only restriction is that it is assumed not to be able to get access to the client's cryptographic keys.

We aim for achieving authenticity in the sense that only contents actually inserted into `sec-cs` by the user can be successfully retrieved, and we aim for confidentiality in the sense that the backend cannot obtain any part of any stored content. Security guarantees are defined formally in conjunction with efficiency goals in Sec. 4.4 due to their interdependence. A general overview is given beforehand.

Note that the model allows a backend to mount DoS attacks (e.g., deleting data). As such attacks are detectable by a user, a backend has a financial incentive in avoiding it. Also, it could be prevented easily via replication.

4.3 Security Concept

Due to its storage efficiency guarantees, `sec-cs` requires a tailored security concept. We discuss reasons and design decisions now and give a formal definition thereafter.

Using cryptographic hash values to reference nodes of a chunk tree yields a Merkle-tree[26]-like data structure that trivially guarantees integrity of a content given the identifier of the root node of its chunk tree. We can use a *message authentication code (MAC)* with a secret, symmetric key instead of an unkeyed hash to also guarantee authenticity of contents stored in the data structure.

Integration of confidentiality is more complicated: For ideal guarantees, we would have to encrypt contents (e.g., using a symmetric block cipher) before constructing their chunk trees as to authenticate their ciphertexts (*encrypt-then-authenticate*). Unfortunately, this would prevent data deduplication: With a randomized encryption scheme, deduplication would not be possible at all, and with a deterministic scheme, deduplication would only be possible at the granularity of complete contents. To allow for storage efficiency, we have to employ encryption at the granularity of the chunks that are to be deduplicated.

A straightforward application of the generally favorable *encrypt-then-authenticate* approach on chunk tree node representations utilizing a randomized encryption scheme, however, would still prevent deduplication as even identical chunk tree nodes yielded different MAC tags (thus different keys) due to different ciphertexts. *Authenticate-then-encrypt* can also be considered secure for specific instantiations [20], but randomized encryption of MAC tags would lead to the same problem. The third option would be *encrypt-and-authenticate*. If applied to chunk tree nodes during insertion into the backend, deduplication would be possible even with randomized encryption, as each chunk tree node was associated a randomized ciphertext during its *first* insertion without affecting other parts of the data structure. Application of encrypt-and-authenticate, however, is generically considered insecure even for practical MAC instantiations[1], thus requiring a careful analysis of the security properties actually achieved by any specific instantiation.[2] [20]

[1]i.e. MAC schemes whose tags do not leak information about inputs

[2]Note that some generic security flaws of encrypt-and-authenticate do not apply in the specific setting at hand as equality of plaintexts is intentionally leaked for the sake of data deduplication, anyway.

To avoid any of these potential pitfalls, we achieve *confidentiality and authenticity* by using an *authenticated, deterministic encryption scheme* to encrypt and authenticate chunk tree nodes before their insertion into the backend. Block cipher modes like EAX [2] and OCB [33, 21] would be suitable for this purpose. They provide confidentiality and authenticity and their ciphertexts are length-preserving (except for the authentication tag), eliminating padding-induced storage overhead. These schemes, however, depend on a nonce that would have to be stored somehow to allow decryption of persisted chunks, which again caused overhead. The SIV construction [34] solves this issue: by using a plaintext's MAC tag as IV for an underlying, conventional IV-based encryption scheme (e.g., CTR mode), it achieves authentication and length-preserving encryption. While SIV depends on a nonce, too, it is resistant to nonce reuse in the sense that no more information than whether two encrypted plaintexts are identical is leaked. [17] As this is leaked *intentionally* in our system to allow deduplication, it is safe to use SIV without a nonce, leading to storage costs identical to those of an authentication-only solution (i.e., overhead equals authentication tag size).

4.4 Formal Definition

The data structure is now described in detail, including its interface, formal goals and internal algorithms.

4.4.1 Interface and Goals

The minimum operation set for a content data structure includes *insertion* and *retrieval*:

- $k \leftarrow \textsc{PutContent}(m)$ shall insert the content m into `sec-cs` and make it accessible by the key k.
  We state the following *storage efficiency goals*:

  G1: The (expected) increase of the data structure's storage consumption caused by $\textsc{PutContent}(m)$ should be in $\mathcal{O}(|m|)$.

  G2: If m is *highly redundant*, i.e., another m' is already stored that is identical to m except for a single sequence of δ bytes, the expected increase in storage consumption caused by $\textsc{PutContent}(m)$ shall be in $\mathcal{O}(\delta + \log |m|)$.

  Note that G2 is defined rather vaguely. Its precise semantics depends on the choice of $\mathcal{C}$: For SC, the difference between contents m and m' is defined as the smallest byte range that would have to be copied from m to m' to turn m' into m, or vice versa. Since CDC supports shifting of contents, some differences between two contents can be represented more compactly, i.e., by a sequence of bytes that is inserted at or removed from a specific byte offset of one content.

The more efficient the underlying chunking scheme, the stronger is thus the goal.

- $m \leftarrow$ GETCONTENT(k) shall retrieve a content m previously inserted into `sec-cs` via the key k.
  We state the following *authenticity goal*:

  G3: If any call $k' \leftarrow$ PUTCONTENT(m') with $k' = k$ has been issued before, then it holds $m \in \{m', \perp\}$.[3]

Goal G2 implies the more general case of m / m' being different in δ bytes spread over x different positions: Imagine the sequence $m' = m_0, m_1, \ldots, m_{x-1}, m_x = m$ of *intermediate* contents with m_i containing the first i differences between m' and m and let δ_i refer to the number of bytes changed between m_{i-1} and m_i. If each of these contents was inserted one after another, insertion of m_i would cause an increase in storage consumption of $\mathcal{O}(\delta_i + \log|m_i|)$, totaling $\mathcal{O}(\sum_{i=1}^{x}(\delta_i + \log|m_i|))$ for all contents. The sizes of $m_1, \ldots, m_x$ are upper-bounded by $|m|$, so total increase in storage consumption is in $\mathcal{O}(\delta + x\log|m|)$. This boundary sublinear in the length of m would not be possible if only leaf chunks were deduplicated, so Goal G2 implies Req. R2.

As any operation execution has to preserve confidentiality of all contents ever stored, we state the *confidentiality goal* independent from a specific operation:

G4: For each content m ever inserted into the data structure, the storage provider must not learn anything beyond (a) its length, (b) chunk boundary positions leaked by $\mathcal{C}$ for target chunk sizes $S, \frac{S^2}{R}, \ldots, \frac{S^h}{R^{h-1}}$, where h is chosen as in Eq. 1 for $n = |m|$, and (c) equality of chunks of m according to the aforementioned chunk boundary positions (w.r.t. all leaf chunks and superchunks ever stored).

Note that constraints G4b and G4c are unavoidable for achieving storage efficiency, as worked out in Sec. 4.3. Thus, strength of Goal G4 is highly dependent on $\mathcal{C}$.

4.4.2 Parameters

The data structure's efficiency can be tuned by setting the following parameter during initialization:

- S is the target chunk size, i.e., the expected size of leaf/superchunks generated by multi-level chunking.

Further, there is an implementation-specific parameter R referring to the storage consumption of chunk references in superchunk representations. We require $S \geq 2R$, which will allow us to meet Goal G1 (see Sec. 5).

[3]This has two important implications: We neither employ measures against malicious deletion of contents nor against rollback attacks, and there are no guarantees that a retrieved content has been inserted before. The reason is that we do not distinguish between *contents* and *chunks* in the data structure, so chunks of contents can be retrieved directly.

4.4.3 Required Algorithms and Assumptions

`sec-cs` is based on some algorithms and assumptions:

- Let $\Pi_E = (\text{GEN}_E, \text{ENCAUTH}, \text{DECVRFY})$ be a DAE-secure (see [34]), deterministic authenticated encryption scheme that generates length-preserving ciphertexts and *message authentication codes (MACs)* of length D. Note that MACs are used to reference chunks, so it holds $R = D$.
- Let $\mathcal{C}$ be a deterministic, *single-level* chunking scheme that produces chunks of a (configurable) expected length S' as used in Sec. 3.
- We assume that the backend's storage costs for storing a key-value pair (k, v) are in $\mathcal{O}(|k| + |v|)$.

4.4.4 Operations

Now we are ready to define the behaviour.

Initialization When `sec-cs` is initialized, parameter S is specified and GEN$_E$ is executed to determine a symmetric cryptographic key K for authenticated encryption.

Content Insertion: $k \leftarrow$ PUTCONTENT(m) Insertion of contents is performed according to the definition of ML-$\mathcal{C}$ (see Sec. 3) which is refined here. First, the height h of the chunk tree t for content m is calculated according to Eq. 1. The tree is then built and its nodes are persisted by executing the recursive Alg. 1, which utilizes $\mathcal{C}$ to perform the appropriate chunking of the content at each individual level and yields some key k' for the root node. We return $k = (k', h)$ as the content's key.[4]

Each node is persisted by the algorithm using PUT. Confidentiality is achieved by encrypting node representations; deduplication+authentication are achieved by using MACs as keys. As superchunks are represented as lists of their children's keys, this yields a Merkle-Tree-like structure of MAC values, achieving authentication of contents.

Algorithm 1 Chunk insertion

```
Precondition: $m'$ is content, $h' \geq 0$ height of to-be-created tree
1:  function PUTCHUNK($m'$, $h'$)
2:      if $h' = 0$ then                                      ▷ create leaf chunk
3:          $c', k' \leftarrow$ ENCAUTH($K$, $m'$)
4:          PUT($k'$, $c'$)
5:      else                                                  ▷ create superchunk
6:          children ← []
7:          Apply $\mathcal{C}$ to $m'$ with target chunk length $\frac{S^{h'}}{R^{h'-1}}$
8:          for all chunks $m''$ produced by $\mathcal{C}$ do           ▷ create children
9:              children.append(PUTCHUNK($m''$, $h' - 1$))
10:         $c', k' \leftarrow$ ENCAUTH($K$, children)
11:         if GET($k'$) = $\perp$ then
12:             PUT($k'$, $c'$)                               ▷ only insert new chunk
13:     return $k'$
```

[4]Inclusion of h is an auxiliary construction. It enables equal length and size for leaf chunks by not requiring storage of the node type.

Content Retrieval: $m \leftarrow$ **GETCONTENT**(k) Retrieval of a content works similar to its insertion. First, the root chunk key k' and the tree's height h' are extracted from the content key k. Afterwards, the recursive Alg. 2 is executed, which retrieves all nodes of the chunk tree and concatenates its leaf chunks, yielding the corresponding content m. Each node is decrypted and checked for authenticity on that way. The algorithm aborts if any node is missing or any node with an erroneous MAC tag is retrieved from the backend. The operation yields $\perp$ then.

Algorithm 2 Chunk retrieval

```
Precondition: $k'$ is chunk key, $h' \geq 0$ the height of its chunk tree
1: function GETCHUNK($k', h'$)
2:     $c' \leftarrow$ GET($k'$)
3:     $v' \leftarrow$ DECVRFY($K, c', k'$)
4:     if DECVRFY failed then Failure                ▷ abort on invalid MAC
5:     return $v'$ if $h' = 0$ else $\|_{k'' \in v'}$ GETCHUNK($k'', h' - 1$)
```

We also designed optimized, non-recursive variants of these operations, which are equivalent but more computationally efficient as they need only a single pass of chunking. They are omitted from the paper due to space restrictions, but included in our implementation (see Sec. 6).

5 Correctness and Security Analysis

We show that the operations from Sec. 4.4.4 achieve the goals from Sec. 4.4.1. Note that the ability of $\mathcal{C}$ to produce chunks of an expected length is crucial for the discussion.

Content Insertion *Insertion* builds a chunk tree whose nodes at each level each represent the whole inserted content. All nodes are persisted in the backend and made accessible by individual keys. As the root node's key transitively allows access to all nodes, the operation is consistent with the required interface. Regarding storage efficiency, we already showed a constant expected per-chunk storage consumption in Sec. 3. Thus, it is sufficient to consider the number of modified chunks to analyze the asymptotic storage costs incurred by the operation.

Goal G1 requiring linear storage costs for a content m is achieved due to the following argument: As size and length are equal for leaf chunks and as there cannot be more than $|m|$ leaf chunks in total, storage costs of all leaf chunks are in $\mathcal{O}(|m|)$. Further, as we have $S \geq 2R$, every superchunk is expected to have at least two children, implying less expected superchunks than leaf chunks. This proves an expected total storage consumption of $\mathcal{O}(|m|)$. Goal G2 is analyzed in detail in Sec. 7, so we only provide an informal argument at this point: As described in Sec. 3, a content differing in one byte from an existing content has storage consumption $\mathcal{O}(\log|m|)$. The main technical difference when δ consecutive bytes differ instead of 1 byte is that those δ bytes might be spread over multiple chunks. Concerning storage costs, this is similar to inserting those δ bytes as a separate content, so the storage overhead is limited to $\mathcal{O}(\delta)$, resulting in a total storage consumption of $\mathcal{O}(\delta + \log|m|)$.

Content Retrieval *Retrieval* retrieves all nodes of a previously built chunk tree and concatenates its leaf chunks, trivially fulfilling the interface. To prove authenticity, we formalize Goal G3 with the *authenticity-breaking game*:

1. The data structure is initialized.
2. An adversary $\mathcal{A}$ is given oracle access to INSERTCONTENT and to the implementation of `sec-cs`. She may issue queries at choice to fill it and is granted full read/write access to the backend.
3. At some point, $\mathcal{A}$ outputs an identifier k.
4. We say $\mathcal{A}$ *wins* if a retrieve query for k returns m' but an insert of a different $m \neq m'$ was performed under identifier k before. Otherwise $\mathcal{A}$ loses.

Using this game, the authenticity property can be shown:

Claim. *If MACs produced by Π_E are unforgeable under a chosen message attack, no adversary can win the auth.-breaking game with non-negligible probability.*

Proof. Assume $\mathcal{A}$ wins the game with non-negligible probability. Let k be the identifier and let m' be the forged content returned by *retrieve*. As the operation only depends on GETCHUNK calls, which in turn only depend on GET operations, at least one GET call during execution of retrieve must have returned a forged result. Let $c' \leftarrow$ GET(k') be the first such call. By definition of Alg. 2, verification of k' being a correct MAC for v' must have been true for retrieve to be successful. Then, as $\mathcal{A}$ knows the algorithms used by the data structure, $\mathcal{A}$ is able to find two different values v, v' with the same MAC k (the value she inserted initially and the forged value). As MACs produced by Π_E are assumed to be unforgeable, this happens only with negligible probability, contradicting our assumption and proving Goal G3. $\square$

Content Confidentiality Goal G4 states that an adversary must not learn anything more about any content m ever stored in the data structure than its length, its chunk boundaries according to the used chunking scheme, and equality relations across all stored chunks. To prove that no more information is leaked by any operation execution, we show that the intentionally leaked information is sufficient for a consistent simulation of any operation.

Let M be the set of contents for which PUTCONTENT is executed at any time, let $m \in M$ be any fixed content and let $\mathcal{A}$ be an adversary trying to obtain information about m. Acc. to Constraint G4a, $\mathcal{A}$ is allowed to know the content's length $n = |m|$. Since the data structure's parameters (see Sec. 4.4.2) are public, $\mathcal{A}$ can, thus, determine the height h of the chunk tree t of m acc. to Eq. 1.

Constraint G4b reveals the chunk boundaries of m output by $\mathcal{C}$ for chunk sizes $S, \frac{S^2}{R}, \ldots, \frac{S^h}{R^{h-1}}$. It is easy to see that these are exactly the chunk boundaries that are computed during a legitimate PUTCONTENT(m) call, i.e., in line 7 of every execution of Alg. 1. In combination with the length of m, $\mathcal{A}$ can, thus, determine the byte ranges of all leaf chunks and superchunks of m. This allows her to construct an abstract chunk tree $\hat{t}$ that has the exact same structure as t, but whose nodes contain *abstract* chunk representations that represent only the respective chunk's length instead of actual chunk representations.

Since equality of any two chunks ever stored is leaked acc. to Constraint G4c, $\mathcal{A}$ can further assign a unique identifier to any (abstract) chunk representation so that the identifiers of two chunk representations are equal iff their represented contents are identical. Without loss of generality, we assume that $\mathcal{A}$ assigns identifiers of the form $\hat{k} = (\hat{k'}, \hat{v})$, where $\hat{k'}$ is a value of length R chosen uniformly at random and $\hat{v}$ is a value chosen uniformly at random whose length equals the represented chunk's length in case of a leaf chunk or $y \cdot R$ in case of a superchunk with y children ($\mathcal{A}$ can calculate these values based on $\hat{t}$).

Now we can show that $\mathcal{A}$ can also be provided with the encrypted/auth. representations of all chunks ever stored without revealing further information about any m.

Claim. *If Π_E is DAE-secure, the probability that an adversary learns anything beyond G4 about any content m from the nodes stored in the data structure is negligible.*

Proof. Assume $\mathcal{A}$ is able to learn something from the encrypted and authenticated chunk representations beyond the aforementioned information with non-negligible probability. First, it is easy to see that the lengths of $\mathcal{A}'s$ previously generated chunk identifiers are equal to the lengths of the actual chunk representations, so she cannot learn anything from the lengths. Being able to learn something from the chunk representations thus implies that she is able to distinguish whether she is given the actual encrypted and authenticated chunk representations or just random strings with the respective lengths.

Let A be her algorithm that on input the information about all contents ever stored in `sec-cs` as stated in G4 and a complete set of chunk representations of the respective lengths outputs 1 if the chunk representations are actual chunk representations and 0 otherwise. Now construct an algorithm B with access to an ENCMAC oracle (with a randomly chosen key) as follows:

1. Initialize a new `sec-cs` data structure and insert all contents $m' \in M$, using the oracle as encryption function, but remembering and reusing oracle outputs instead of asking for same input multiple times.
2. Pass all information about every content of M as stated in G4 as well as all (encrypted and authenticated) chunk representations to A, yielding output x.
3. Return x.

If B has access to an actual ENCMAC oracle, A is given actual chunk representations as created by the data structure. If a random oracle \$ with outputs of respective lengths is given to B instead, A gets only random data. If A is able to distinguish both cases with non-negligible probability, B is thus able to distinguish a random oracle from an encryption oracle with non-negligible probability. According to the definitions given in [34], B would be an adversary with non-negligible DAE-advantage, which contradicts the assumption that Π_E is DAE-secure. □

At this point, $\mathcal{A}$ knows the complete chunk tree t for every content m ever stored in a `sec-cs` instance, including the (encrypted and authenticated) chunk representation of every chunk tree node. We have already seen that $\mathcal{A}$ cannot obtain more information about any content based on this data than stated in Goal G4. Now we show that even metadata (i.e., access patterns from individual operation executions) do not reveal anything more about any individual content. The idea of the proof is as follows: When a data structure operation is executed, $\mathcal{A}$ can only see KVS operation calls made by `sec-cs`. If $\mathcal{A}$ is able to simulate any data structure operation execution to the extent that all KVS operation calls are consistent to a real execution based on information she already has, she does not learn anything from a real operation execution.

Consider a PUTCONTENT(m) call. Its execution simply consists of a call of Alg. 1 with an additional argument h. Knowledge of h allows her to simulate that call, although she cannot provide the content m to Alg. 1. The algorithm can be executed consistently given t, though: Consider any execution of PUTCHUNK(m', h'). If $h' = 0$, the execution corresponds to a leaf chunk of t that is encrypted, authenticated and inserted using PUT. Since $\mathcal{A}$ already knows the representation of the corresponding chunk, she can simply issue the PUT call of line 4. Otherwise, Alg. 1 performs chunking on the respective chunk, issues recursive calls for the resulting chunks and inserts an encrypted, authenticated superchunk. $\mathcal{A}$ can perform the recursive calls by extracting the children of the current chunk from t; she can determine the superchunk representation c', k' from line 10 as it is contained in t, and she can issue the GET and PUT calls from lines 11–12 since they depend only on k', c' and the children's identifiers.

The call of Alg. 2 during GETCONTENT(k) is possible for $\mathcal{A}$ due to the same reasons as before. Each individual recursive GETCONTENT call corresponding to a node of t can be trivially simulated by $\mathcal{A}$: The only operation not directly executable by $\mathcal{A}$ is the DECVRFY call in line 3. For the simulation, though, it is sufficient to distinguish three cases. First, DECVRFY fails whenever k' is not a valid authentication tag corresponding to ciphertext c'. Since $\mathcal{A}$ knows the correct chunk representation k'', c'' for the respective chunk from t, she can assume the call to be successful iff $c'' = c'$ except with negligible probability. Second, if $h' > 0$, k' is the identifier of a superchunk, so v' is a list of its children's keys, which she can simply extract from t. Only if $h' = 0$, $\mathcal{A}$ fails to compute v'. In this case, however, all subsequent operations performed by a benign client are exclusively local (without any feedback to the storage backend), so the simulation is consistent and sound from an adversary's perspective.

Thus, $\mathcal{A}$ is able to perform consistent simulations of all operations, which proves Goal G4.

Note that choice of $\mathcal{C}$ defines a trade-off between confidentiality and storage efficiency. If $\mathcal{C}$, e.g., was WFC, strongest security guarantees could be achieved (although this would fail to achieve storage efficiency): G4b would not leak any information at all and G4c would only leak equality of complete contents. If SC was used, G4b would still not leak any information as its output depends only on a content's length which is covered by G4a, but equality of (small) chunks naturally provides an adversary with more information. For CDC schemes, after all, G4b becomes relevant as chunk boundaries are computed based on plaintext content parts. Precise security implications depend on the specific scheme and cannot be determined in general. An analysis for one scheme is given in [22].

6 Implementation

To ease adoption in practice and to allow for an empirical evaluation (see Sec. 7), we have created an implementation. The data structure including our chunking scheme ML-* is wrapped into a flexible Python module named `seccs` available for download in the Python Package Index [29] or via `pip install seccs`. Unit tests verifying the implementation's correctness w.r.t. Goals G1, G2 and G3 are bundled with the module.

Since we could not find a sufficiently efficient rolling hash Python implementation, we also developed a rolling-hash-based chunking module `fastchunking` compatible to `seccs` and available for download in PyPI as well. It is a wrapper for parts of the highly efficient *ngramhashing* C++ library [40] by Daniel Lemire and thus able to outperform pure-Python implementations.

7 Evaluation

We present an extensive evaluation of `sec-cs`'s storage efficiency consisting of two parts: Sec. 7.2 confirms our stated efficiency goals both analytically and empirically and Sec. 7.3 compares the performance of `sec-cs`'s novel chunking scheme ML-* to other approaches.

7.1 Assumptions

To provide concrete numbers, we make some assumptions about the implementation of `sec-cs`. We assume that a deterministic authenticated encryption scheme with length-preserving ciphertexts and $(D = 32)$-bytes MACs is used (e.g. AES-SIV-256), resulting in a constant storage requirement of $R = D = 32$ bytes for chunk references.

By *storage costs*, we refer to the storage consumption of the used KVS (see Sec. 4.1) for some state. To be independent of any specific backend data structure, we ignore any overheads and roughly estimate storage costs as the sum of the sizes of the KVS's elements, where an element's size is the sum of the sizes of its key and value.

7.2 Storage Performance of sec-cs

To complement the proofs from Sec. 5, we evaluate G2 in detail. We do not continue an asymptotic discussion but work out concrete storage costs to judge suitability of `sec-cs` in practice. The analytical deduction is given below and its validity is confirmed empirically afterwards.

7.2.1 Analytical Evaluation

Let m' be a content consisting of random bytes that is already present in `sec-cs`. Goal G2 states that insertion of m differing only in a sequence of δ bytes should cause storage costs in $\mathcal{O}(\delta + \log|m|)$. To work these costs out more precisely, we analyze the border cases first.

Border Case: $\delta = |m|$ If $\delta = |m|$, chunk trees t and t' of m and m' are not expected to share any nodes, so storage of m should cause costs in $\mathcal{O}(|m|)$ acc. to the stated goal. (Note that this case also covers Goal G1.) Since leaf chunks have average length S, the expected number of leaf nodes of t is $\frac{|m|}{S}$. For the same reasons as in the proof in Sec. 5 ($S > 2R$ implies superchunks are expected to have ≥ 2 children), t has more expected leaf than superchunk nodes, so its total expected number of nodes is:

$$\text{ExpN}^{\mathcal{C}}_{|m|} \quad \leq \quad \left\lceil \frac{2 \cdot |m|}{S} \right\rceil \tag{2}$$

Every chunk tree node has an expected size of S according to Sec. 3 and is stored under a D-byte digest, so storage costs for this case are as follows:

$$\text{Storage}^{\mathcal{C}}_{|m|} \quad = \quad (D + S) \cdot \text{ExpN}^{\mathcal{C}}_{|m|} \tag{3}$$

Border Case: $\delta = 1$ If $\delta = 1$, m and m' differ only in 1 byte. Here, storage costs depend on ML-*'s underlying chunking strategy $\mathcal{C}$. In case of $\mathcal{C} =$ SC, t and t' differ in exactly one node at each level, each having size $\leq S$, which trivially results in the following storage costs:

$$\text{AddStrg}^{\text{SC}}_{h} \leq (D + S)(h + 1) \tag{4}$$

For $\mathcal{C} = \text{CDC}$, however, the situation is more complex. First, $\delta = 1$ allows for shifting in case of CDC. Second, t and t' might differ in more than one node at each level. The reason is that the modification of a single byte might change up to W chunk boundaries at each level of the chunk tree, probably causing extra chunks to be inserted as well. To determine the total average number of chunk tree nodes that are inserted in this case, we analyze how many new chunks are created at each chunking level.

Let us fix some height $h', 0 \leq h' \leq h$. At height $h' = h$ we only have a single (root) chunk that represents the whole content m. At height $h' < h$ we deal with chunks of average length $\frac{S^{h'+1}}{R^{h'}}$ (a position is a boundary with probability $\frac{R^{h'}}{S^{h'+1}}$) according to Sec. 3. When considering a fixed W-byte window containing the changed byte, the probability that this window yields a chunk boundary in m but did not yield a chunk boundary in m' is $\left(1 - \frac{R^{h'}}{S^{h'+1}}\right) \cdot \left(\frac{R^{h'}}{S^{h'+1}}\right)$. The same probability holds for the case in which a chunk boundary present in m' is not present in m anymore. As the resulting new chunks and the chunk that has to be inserted anyway at this level are not necessarily consecutive[5], creation (omission) of a chunk boundary in contrast to m' might cause up to 1 (2) additional chunks at height h', respectively.

Since there are up to W window contents containing the changed byte at each chunking level and each position yields 1 or 2 new chunks each with probability $\left(1 - \frac{R^{h'}}{S^{h'+1}}\right) \cdot \left(\frac{R^{h'}}{S^{h'+1}}\right)$, we expect up to $1 + (1 + 2)W\left(1 - \frac{R^{h'}}{S^{h'+1}}\right) \cdot \left(\frac{R^{h'}}{S^{h'+1}}\right)$ new chunks at height h'. As we have a single changed chunk at height h, we get the following upper bound for the expected number of chunk tree nodes differing between t and t':

$$\textsc{ExpNN}_h^{\text{CDC}} \leq 1 + \sum_{h'=0}^{h-1} \left(1 + 3W\left(1 - \frac{R^{h'}}{S^{h'+1}}\right)\frac{R^{h'}}{S^{h'+1}}\right) \quad (5)$$

While chunking is performed in a way that achieves an average size of S for every chunk when applied to random content, we cannot assume an average size of S for *additional* chunks created for inserting m in addition to m'. The rationale is that new chunks are created from existing chunks *not* chosen uniformly at random: A random position in a content is more likely to hit large chunks than smaller ones as more positions are covered by them.

Consider the chunks of m' at some fixed height h'. As each byte position is a height-h' chunk boundary with probability $p = \frac{R^{h'}}{S^{h'+1}}$, the probability for a chunk having length c is $(1-p)^{c-1} \cdot p$. Note that $0 < p < 1$ since $S > R$. As we expect an average number of $|m'|/\frac{1}{p}$ chunks at height h' in total, the expected number of length-c chunks at height h' is $|m'|p \cdot (1-p)^{c-1} \cdot p$. Since each of those chunks covers c bytes and since the total content length is $|m'|$, the fraction of the content that is covered by chunks of length c is:

$$\left(\frac{c}{|m'|}\right) \cdot |m'|p \cdot (1-p)^{c-1} \cdot p \quad = \quad cp^2 \cdot (1-p)^{c-1}$$

Thus, the expected *length* of a height-h' chunk at a position chosen uniformly at random is:

$$\begin{aligned}
&\sum_{c=1}^{|m'|} c^2 p^2 \cdot (1-p)^{c-1} \;=_{p<1}\; \frac{p^2}{1-p}\sum_{c=1}^{|m'|} c^2 \cdot (1-p)^c \\
\leq\; &\frac{p^2}{1-p}\sum_{c=1}^{\infty} c^2 \cdot (1-p)^c \;=_{|1-p|<1}\; \left(\frac{p^2}{1-p}\right)\left(\frac{p^2-3p+2}{p^3}\right) \\
=\; &\frac{p^2-3p+2}{p(1-p)} \;=\; \frac{2}{p} - 1 \;=\; 2\left(\frac{S^{h'+1}}{R^{h'}}\right) - 1
\end{aligned}$$

As height-h' superchunks store R-byte references to height-$(h'-1)$ chunks of avg. length $\frac{S^{h'}}{R^{h'-1}}$ and as size equals length for leaf chunks, the exp. *size* of a height-h' chunk at a random position (and thus the exp. size of any chunk created when inserting m) is upper-bounded by:

$$\textsc{ExpChunkSize}^{\text{CDC}} \quad \leq \quad 2 \cdot S \quad (6)$$

This results in the following storage requirement:

$$\textsc{AddStrg}_h^{\text{CDC}} \leq (D + 2S)\,\textsc{ExpNN}_h^{\text{CDC}} \quad (7)$$

Remaining Case: $1 < \delta < |m|$ In the remaining case, i.e., m differing from m' in a sequence of more than 1 but less than $|m|$ bytes, both the first and the last byte of the change bytestring affect chunks as in the case $\delta = 1$. For ML-CDC, they are likely to be part of *large* chunks of expected length $2S$ for the same reasons as discussed above. We conservatively estimate that these two bytes cause storage costs of $2 \cdot \textsc{AddStrg}_h^{\mathcal{C}}$ (with $h = \left\lceil \log_{\frac{|m|}{S}}\left(\frac{S}{R}\right)\right\rceil$). The remaining bytes are either part of those chunks (so their storage costs have already been accounted for), or they result in the same chunks that would be created if they were inserted into `sec-cs` as a separate content, causing storage costs up to $\textsc{Storage}_{\delta-2}^{\mathcal{C}}$. Thus, we estimate the total storage requirement as follows:

$$\textsc{DeltaStrg}_{h,\delta}^{\mathcal{C}} \quad \leq \quad 2 \cdot \textsc{AddStrg}_h^{\mathcal{C}} + \textsc{Storage}_{\delta-2}^{\mathcal{C}} \quad (8)$$

As h is logarithmic in $|m|$, all cases fulfill Goal G2.

7.2.2 Empirical Evaluation

As there are no conceptual differences between ML-SC / ML-CDC w.r.t. to the goal evaluated in this section, we focus on the more interesting ML-CDC scheme in the empirical evaluation. We perform experiments with our implementation (see Sec. 6) meeting the parameters from

[5]Changing a single byte might change up to W boundaries in an area of W bytes after the change position, but only the chunk before the first boundary contains the change. If two consecutive boundaries in this area exist in m' and m, the chunk in between is unchanged, but might be followed by changed chunks if further boundaries are changed.

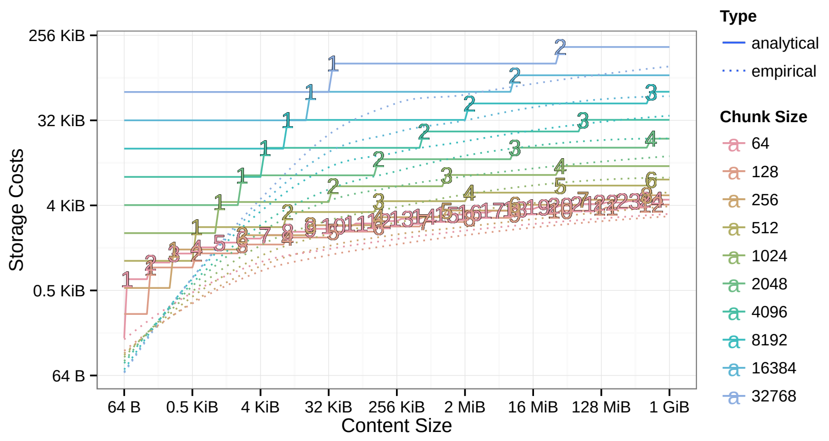

Figure 1: Storage costs for modified contents ($\delta = 1$)

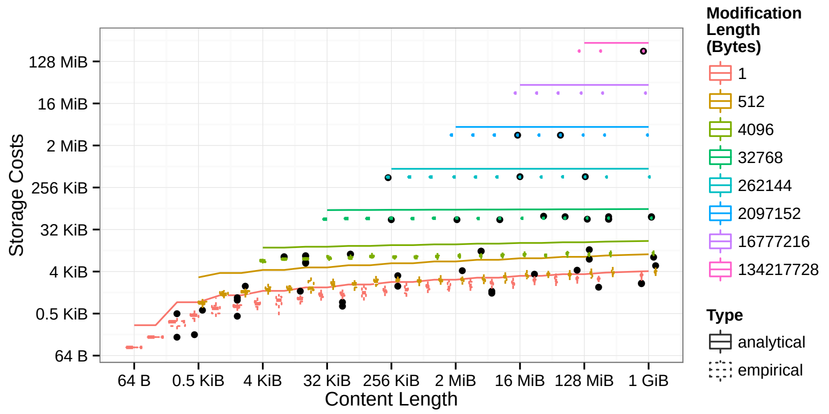

Figure 2: Storage costs for different modifications ($S = 128$)

Sec. 7.1, using a Rabin-Karp-based CDC scheme with window size $W = 48$ bytes (based on evaluation results of the CDC authors [28]) and no min/max chunk size set.

We simulate the scenario from the analytical evaluation: We choose a content m of size $|m|$ uniformly at random, insert it into an empty `sec-cs` instance and remember its state. We replace a randomly chosen δ-bytes substring ($1 \leq \delta \leq |m|$) of m by a different δ-bytes substring chosen uniformly at random, insert the resulting content m' and compare `sec-cs`'s size to the remembered one to measure increase in storage costs. We executed the experiment 20 times for each combination of content size $|m|$, chunk size S and δ, including border cases $\{1, |m|\}$.

Fig. 1 compares empirical and analytical results for $\delta = 1$, showing the minimal expected storage overhead for insertion of highly redundant contents. Solid lines show the calculated relation between content sizes and increase in storage costs for different chunk sizes. While sublinear growth is visible for either chunk size, smaller sizes result in even lower storage costs, unless the chunk size is chosen unreasonably small: For the smallest evaluated chunk size (64 bytes), costs incurred by additional superchunk levels outweigh the smaller per-chunk costs.

Threshold content sizes resulting in a respective number of chunking levels are indicated by the positions of the numbers in the chart. The reason for the leaps at threshold sizes is that our estimate is based on a constant chunk size, while root nodes are smaller in practice (proportional to content size for a fixed tree height), resulting in smaller trees. Empirical results confirm growth is smoother in practice: Dotted lines show LOESS [11] curves fitted to the measured increase in storage costs (smoothing parameter 0.75, degree 2), summarizing its relation to content length for the respective chunk sizes. Results are in line with the calculated upper bounds, confirming sublinear growth in general and least overhead for $S = 128$.

To verify whether the most promising chunk size of $S = 128$ bytes also yields small overheads for larger modifications, results for $S = 128$ in the general case ($\delta \geq 1$) are shown in detail in Fig. 2. Solid lines represent analytical upper bounds for different modification lengths, yielding least costs for $\delta = 1$ (red). Empirical data including outliers (black points) are illustrated as box plots, which are barely visible since they indicate rather small fluctuations in storage costs. The box plots confirm that the analytical bound is conservative, especially for $\delta > 1$: Even outliers are below their corresponding analytical line.

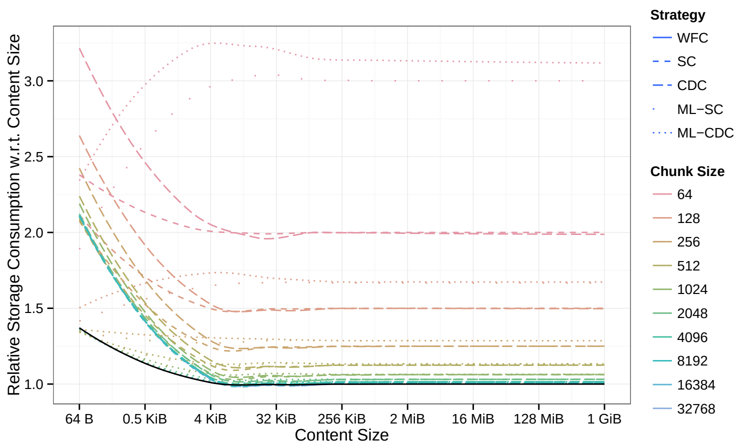

Figure 3: Storage costs relative to content size

7.3 Comparison of WFC / SC / CDC / ML-*

We compare performance of ML-* empirically to that of the other approaches. To allow a fair comparison, we evaluate all schemes with `sec-cs`. Note that all schemes are in fact special cases of ML-SC / ML-CDC: Fixing the height of generated chunk trees to 1 results in SC / CDC, respectively, where all metadata representing a content are collected in a single root node; a fixed height of 0 maps each content to a leaf-only tree, corresponding to WFC.

7.3.1 No-Deduplication Storage Overhead

Every scheme incurs storage overhead that is necessary to enable data deduplication. In case of WFC, only a small digest (i.e., a hash value) has to be stored for each content for this purpose, resulting in constant overhead. When smaller chunks are stored, overhead is incurred both due to the digests for individual chunks and for storage of a content's representation, i.e., a list of digests of chunks needed to reconstruct it. This overhead is usually compensated by savings from storage of deduplicable contents.

Before comparing the savings achieved by the different schemes, we take a look at the overhead that is incurred when *non-deduplicable* contents are stored. We instantiate `sec-cs` for each chunking strategy and with different chunk sizes and perform the following experiment: We insert a fixed-size content chosen uniformly at random into the empty data structure and measure the *storage expansion factor*, i.e., total storage costs divided by the actual content size. We repeat the experiment for different content sizes, 20 times for each combination of parameters.

The LOESS curves in Fig. 3 show the measured expansion factor for different content sizes, which is constant for larger content sizes. It shows that the expansion factor is nearly 1 (in fact, storage overhead is constant: 32 bytes per content) for WFC (solid black line) and slightly above 1 for any scheme with small chunk sizes. With higher chunk sizes, the overhead grows significantly: SC/CDC (dotted lines) have an expansion factor of 1.25 / 1.5 / 2.0 for chunk sizes 64 / 128 / 256; the expansion factor for ML-* (dashed lines) is even higher due to additional storage of non-root superchunk nodes. Interestingly, ML-CDC produces even more storage overhead than ML-SC. This is due to the concept of ML-* handling each chunk as individual content. In case of ML-SC, the last nodes at each chunk tree height represent chunks smaller than S, causing overproportional costs, while any other node is the root of a full, balanced tree. In ML-CDC, these costs are caused recursively by the last nodes of *every* subtree.

7.3.2 Deduplication Overhead without Shifting

To measure possible savings from deduplication, we modify the previous experiment: We insert a second content that differs from the first in only a single byte at an offset chosen uniformly at random. Fig. 4 shows the total increase in storage costs after the second content has been inserted: For WFC, increase corresponds to the inserted content's size; for SC/CDC, storage costs are only a fraction of that thanks to deduplication, but still linear in the content size. Costs of ML-SC/ML-CDC are orders of magnitude lower and sublinear in the content size.

7.3.3 Deduplication Overhead with Shifting

To account for the strengths of CDC, we perform a slight modification of the previous experiment: Instead of overwriting, we *insert* a random byte at a random position, leading to a shift of the remaining content. Results are shown in Fig. 5: As expected, performance of WFC, CDC and ML-CDC is comparable to the previous experiment since they are robust against shifting. SC and ML-SC, however, yield storage costs of about half of the content size for chunk sizes ≥ 256 (with slight variations in chunk sizes), which corresponds to an expected amount of 50% of the content being *before* the shift position and thus deduplicable. Observe that costs for SC with $S = 64$ are similar to WFC, due to storage expansion factor 2.

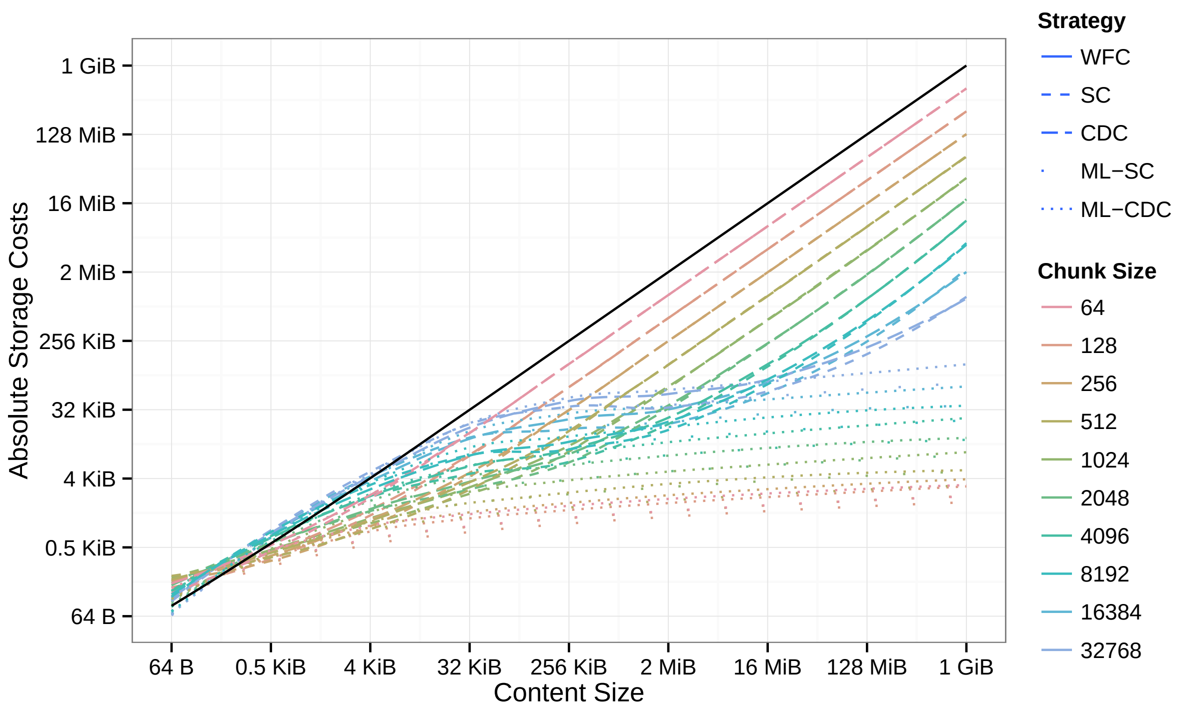

Figure 4: Storage costs for modified content (overwrite)

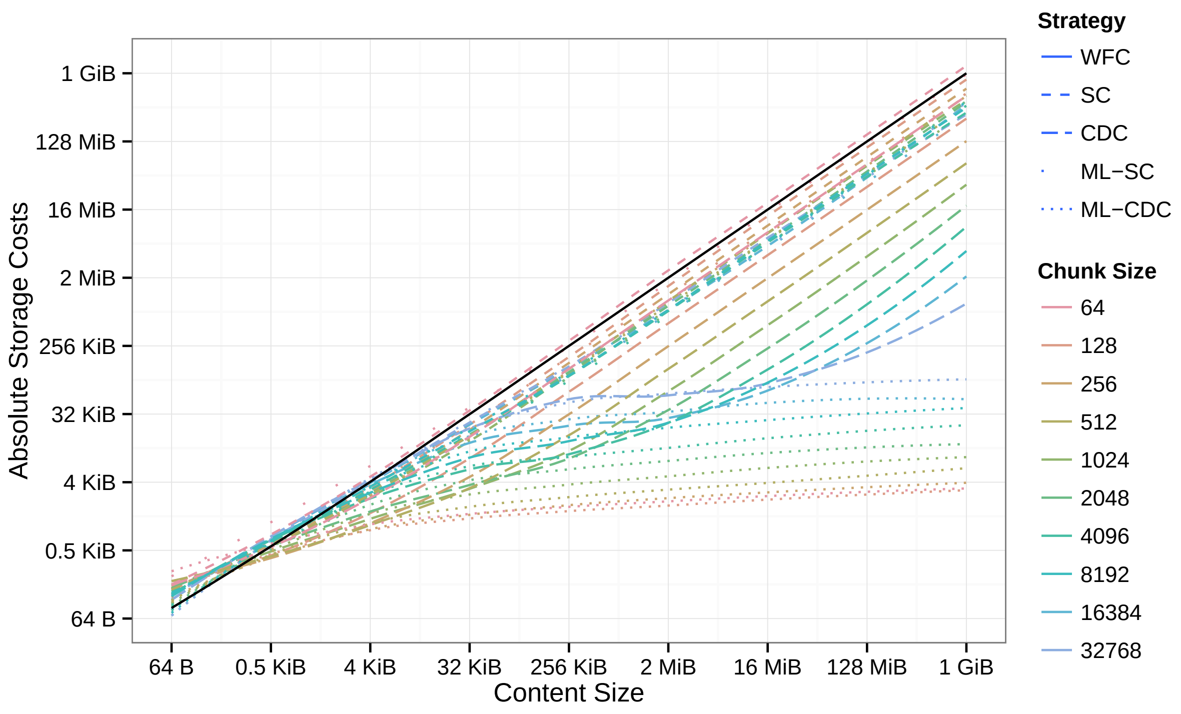

Figure 5: Storage costs for modified content (insert)

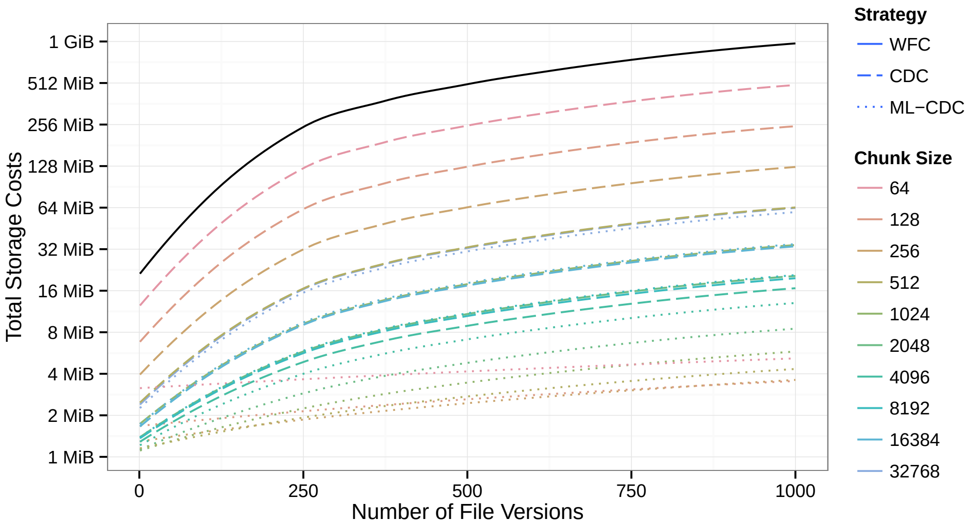

Figure 6: Storage costs for many similar 1 MiB contents

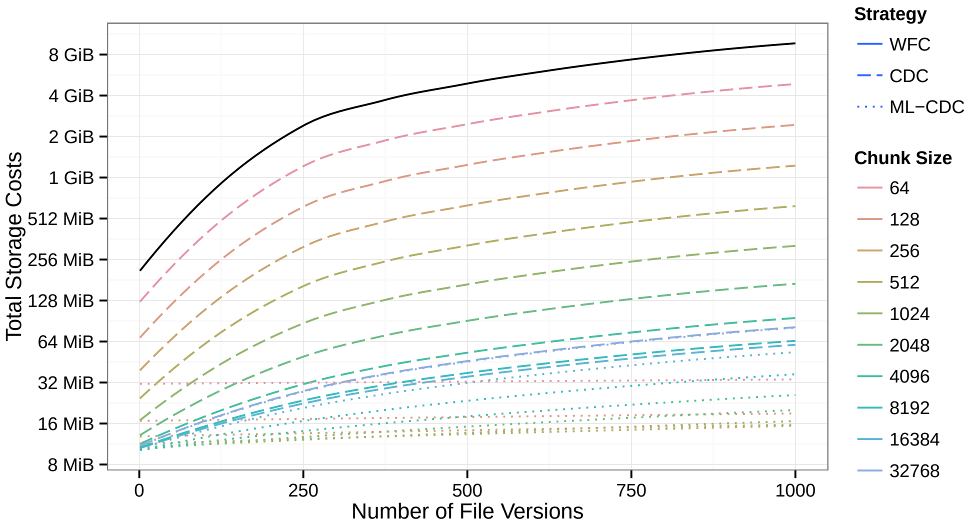

Figure 7: Storage costs for many similar 10 MiB contents

7.3.4 Break-Even Analysis

Sec. 7.3.2 and 7.3.3 have shown that ML-* is significantly more efficient than today's common deduplication strategies when storing contents that differ only slightly from already stored ones, especially for small chunk sizes. However, Sec. 7.3.1 has shown that this efficiency comes at the cost of a higher storage expansion factor, i.e., storage of non-deduplicable contents is more expensive in presence of ML-* and small chunk sizes. This raises the question as to whether and when ML-* is preferable. Intuitively, this is the case whenever storage of *many* versions of contents is involved, e.g., in a backup scenario. We investigate this as follows: We start with a fresh `sec-cs` instance containing a single, random 1 (10) MiB content. Then we insert modifications of this content and measure storage costs after each inserted version.

Fig. 6 (Fig. 7) shows LOESS curves displaying smoothed results over 20 runs for each combination of parameters: As expected, WFC yields storage costs of about 1 MiB (10 MiB) for every stored content version. Costs for CDC are only a fraction thanks to deduplication: $2048 \leq S \leq 8192$ ($8192 \leq S \leq 16384$) yields lowest costs; for other chunk sizes, CDC incurs significantly higher costs. If only few versions are stored, ML-CDC yields slightly lower costs for *any* chunk size between 256 and 8192 (32768) bytes. The more content versions are stored, the more significant are the savings by ML-CDC: When 125 (250) similar versions are stored, ML-CDC with $256 \leq S \leq 1024$ ($256 \leq S \leq 4096$) requires only half of the storage space as the most-efficient CDC variant; for 1000 versions, costs are orders of magnitude lower. Note that we omitted results for SC / ML-SC for readability: Due to shifting, they are close to WFC.

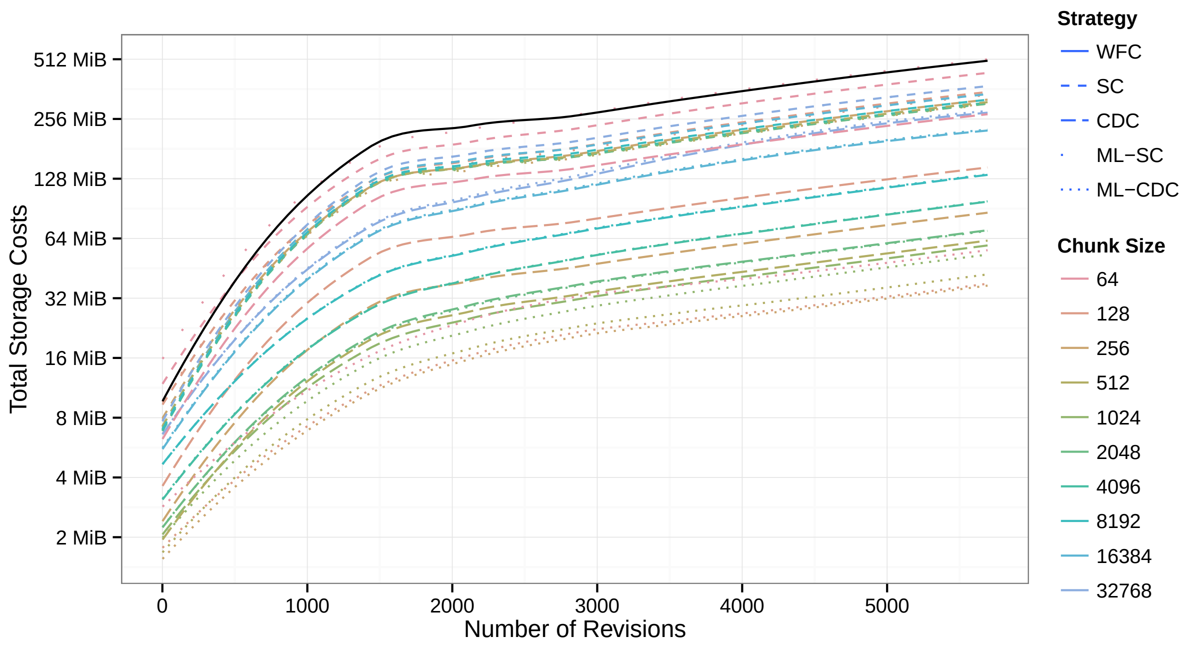

Figure 8: Storage costs for revisions of Redis git repository

7.3.5 Real-Life Data Comparison

While the previous experiments have proven ML-*'s superiority in hypothetical scenarios involving slight changes on random data, it remains to be analyzed how good it performs in real life. We leave an extensive evaluation (e.g., involving a backup system) for future work, but perform the following experiment as a starting point: We insert content versions into `sec−cs` as before, but instead of random data, we insert *all* file contents of all revisions (as of 2016-05-16) of the Redis key-value database git repository [32] and measure `sec−cs`'s storage costs.

Results (Fig. 8) are promising: With about 512 MiB for all 5693 revisions, WFC causes by far the highest costs. SC / ML-SC can only slightly reduce these costs as they are not robust against shifting. Costs for CDC are lower and range from about 64 MiB for $512 \leq S \leq 2048$ to 256 MiB for $S = 64$. For $S \geq 2048$, performance of ML-CDC is comparable to CDC as only a single chunking level is used for most files acc. to Eq. 1. For smaller S, ML-CDC is significantly more efficient than the other schemes. ML-CDC with $128 \leq S \leq 256$ performed best, causing only about 38 MiB of total storage costs, which is rather close to the 21 MiB required by (unencrypted) git.[6]

8 Conclusion

We have introduced a data structure for encrypted and authenticated storage of file contents, `sec−cs`, that employs a novel multi-level chunking strategy, ML-*, to achieve storage efficiency. The data structure transparently deduplicates identical parts of file contents without relying on information about relations between them, and achieves storage costs for highly redundant contents logarithmic in their lengths. We have proven its security and evaluated efficiency extensively w.r.t. other common deduplication concepts. A ready-to-use, open source Python implementation has been published as part of our work as to allow integration in other software projects.

As next step, we work on a backup system based on `sec−cs` and on an extension that supports partial read and write access to contents, making it suitable as backend for future file systems based on untrusted cloud storage.

[6]Note that comparison between git and `sec−cs` is unfair: Git accepts additional computational overhead by computing deltas across contents and it applies compression to aggregated contents which is only possible since it does not support encryption, unlike `sec−cs`.

References

[1] G. Appaji Nag Yasa and P. C. Nagesh. Space savings and design considerations in variable length deduplication. *SIGOPS Oper. Syst. Rev.*, 46(3):57–64, Dec. 2012.

[2] M. Bellare, P. Rogaway, and D. Wagner. The EAX mode of operation. In *Fast Software Encryption – FSE 2004*, pages 389–407, 2004.

[3] https://www.boxcryptor.com.

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

[5] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. Msr-tr-2002-30, Microsoft Research, 2002.

[6] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *USENIX ATC, General Track*, pages 113–126, 2003.

[7] https://www.dropbox.com/.

[8] http://duplicity.nongnu.org/.

[9] https://github.com/vgough/encfs.

[10] K. Eshghi and H. K. Tang. A framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR*, 30, 2005.

[11] J. Fox. Nonparametric regression. Appendix to An R and S-PLUS Companion to Applied Regression, Jan. 2002.

[12] https://git-scm.com/.

[13] https://www.gnupg.org/.

[14] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Proc. NDSS*, 2003.

[15] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *Information Security*, volume 5222 of *LNCS*, pages 80–96. Springer Berlin Heidelberg, 2008.

[16] M. T. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. Technical report, Johns Hopkins Information Security Institute, 2001.

[17] D. Harkins. Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES). RFC 5297 (Informational), Oct. 2008.

[18] A. Heitzmann, B. Palazzi, C. Papamanthou, and R. Tamassia. Efficient integrity checking of untrusted network storage. In *Proc. StorageSS '08*, pages 43–54, New York, NY, USA, 2008. ACM.

[19] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proc. USENIX FAST*, pages 29–42, 2003.

[20] H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In *Proc. Crypto 2001*, pages 310–331, 2001.

[21] T. Krovetz and P. Rogaway. The software performance of authenticated-encryption modes. In *Fast Software Encryption – FSE 2011*, pages 306–327, 2011.

[22] D. Leibenger and C. Sorge. A storage-efficient cryptography-based access control solution for subversion. In *Proc. SACMAT '13*, 2013.

[23] C. Liu, Y. Lu, C. Shi, G. Lu, D. Du, and D.-S. Wang. Admad: Application-driven metadata aware de-duplication archival storage system. In *Proc. 5th IEEE Int. Workshop on Storage Network Architecture and Parallel I/Os*, pages 29–35, 2008.

[24] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[25] D. Meister and A. Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *Proc. SYSTOR 2009*, pages 8:1–8:12.

[26] R. C. Merkle. A certified digital signature. In *Proc. CRYPTO '89*, pages 218–238. Springer New York, 1990.

[27] A. Miller, M. Hicks, J. Katz, and E. Shi. Authenticated data structures, generically. In *Proc. POPL '14*, pages 411–423, 2014.

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

[29] https://pypi.python.org/pypi/.

[30] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *Proc. 1st USENIX FAST*, 2002.

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

[32] https://github.com/antirez/redis.git.

[33] P. Rogaway, M. Bellare, J. Black, and T. Krovetz. OCB: A block-cipher mode of operation for efficient authenticated encryption. In *Proc. ACM CCS '01*, pages 196–205, 2001.

[34] P. Rogaway and T. Shrimpton. A provable-security treatment of the key-wrap problem. In *Proc. EUROCRYPT '06*, pages 373–390, 2006.

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

[36] R. Tamassia. Authenticated data structures. In *Algorithms - ESA 2003*, volume 2832 of *LNCS*, pages 2–5. Springer Berlin Heidelberg, 2003.

[37] D. Teodosiu, N. Bjorner, J. Porkka, M. Manasse, and Y. Gurevich. Optimizing file replication over limited-bandwidth networks using remote differential compression. Technical Report MSR-TR-2006-157, Microsoft Research, November 2006.

[38] The Apache Software Foundation. Apache Subversion. `http://subversion.apache.org/`, Apr. 2015.

[39] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Department of Computer Science, Australian National University, June 1996.

[40] https://github.com/lemire/rollinghashcpp.

[41] M. Vrable, S. Savage, and G. M. Voelker. Cumulus: Filesystem backup to the cloud. *Trans. Storage*, 5(4):14:1–14:28, Dec. 2009.

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