Bonsai: A Generalized Look at Dual Deduplication

Hadi Sehat\textsuperscript{1}, Anders Lindskov Kloborg\textsuperscript{2}, Christian Morup\textsuperscript{2}, Elena Pagnin\textsuperscript{2}, Daniel E. Lucani\textsuperscript{1}

\textsuperscript{1}Agile Cloud Lab, Department of Engineering, DIGIT, Aarhus University, Aarhus, Denmark
\textsuperscript{2}Department of Engineering, Aarhus University, Aarhus, Denmark
\textsuperscript{3}Department of Electrical and Information Technology, Lund University, Lund, Sweden
\{hadi,daniel.lucani\}@eng.au.dk, \{andersklborg, Chmorup\}@gmail.com, elena.pagnin@eit.lth.se

Abstract—Cloud Service Providers (CSPs) offer a vast amount of storage space at competitive prices to cope with the growing demand for digital data storage. Dual deduplication is a recent framework designed to improve data compression on the CSP while keeping clients’ data private from the CSP. To achieve this, clients perform lightweight information-theoretic transformations to their data prior to upload. We investigate the effectiveness of dual deduplication, and propose an improvement for the existing state-of-the-art method. We name our proposal Bonsai as it aims at reducing storage fingerprint and improving scalability. In detail, Bonsai achieves (1) significant reduction in client storage, (2) reduction in total required storage (client + CSP), and (3) reducing the deduplication time on the CSP. Our experiments show that Bonsai achieves compression rates of 68\% on the cloud and 5\% on the client, while allowing the cloud to identify deduplications in a time-efficient manner. We also show that combining our method with universal compressors in the cloud, e.g., Brotli, can yield better overall compression on the data compared to only applying the universal compressor or plain Bonsai. Finally, we show that Bonsai and its variants provide sufficient privacy against an honest-but-curious CPS that knows the distribution of the Clients’ original data.

Index Terms—Private Cloud Storage, Compression, Privacy, Secure Deduplication

I. INTRODUCTION

Due to emerging applications that produce massive amounts of data, many systems choose to use Cloud Service Providers (CSPs) to outsource the storage of their data. In order to deliver a cost-effective service to their clients, CSPs tend to use compression techniques to reduce the fingerprint of the data. This includes a number of file compression techniques, e.g., DEFLATE [1], BZIP2 [2], Brotli [3], as well as delta encoding approaches, e.g., VCDIFF [4], which focus on individual chunks of data or files for compression. As an alternative, data deduplication [5] removes duplicate data across different files and even different users in the system. Data deduplication stores a copy of each unique chunk and represents files with pointers to these unique chunks. In a nutshell, it avoids storing duplicates [6]. In most practical storage systems, both deduplication and compression are used for a more efficient reduction of the fingerprint of the data [7].

Recently, generalized deduplication [8] has been proposed as an improvement to data deduplication. In this case, the storage system is not only capable of removing exact duplicates, but also chunks of files that are close to each other according to a certain metric [9]. At its core, generalized deduplication in a cloud setting works as follows. The CSP maintains a set of bases. For each received chunk of file, if the chunk is already present in the set of bases the CSP duplicates it by storing a file identifier and a pointer representing the base. Otherwise, the CSP looks for a base similar to the given chunk, i.e., a chunk in the set of bases that has a short distance to the received chunk, e.g., according to the Hamming distance [9] or the Swap distance [10]. If such a base is found, the CSP deduplicates this record in a generalized way by storing a file identifier, a pointer representing the closest base, and a string containing the difference between the base and the chunk. Finally, if no similar bases are present in the set of stored bases, the CSP considers the chunk as a new base to add to the list of bases.

A major drawback of allowing the CSP to carry out (generalized) deduplication is that the CSP must be provided with the actual data. This is not desirable in applications where clients may upload sensitive information or are concerned about data privacy. A naive solution would be to encrypt the data and upload only ciphertexts. This approach, however, drastically reduces the deduplication capability on the CSP, as secure encryption implies little or no correlation among ciphertexts. An alternative approach is to use convergence encryption [11] to maintain a balance between privacy and compression. The main weakness of convergence encryption is that a nosy CSP or even a malicious client can attack the system to discover which chunks have been stored previously in the system, exposing sensitive information to the malicious party. Moreover, the CSP can obtain exact information about the files of each user given sufficient time and resources, by breaking the encryption of the outsourced data [12].

The work by Vestergaard et al. [13] proposes an alternative mechanism to convergence encryption, which is called Multi-Key revealing Encryption (MKRE). In [13] chunks are transformed before upload so that any two similar chunks yield outsourced data that are close to each other. Although this method is shown to be more robust against attacks to break the encryption, its security is proven only in the programmable random oracle model, which is not realistic as it cannot be faithfully implemented by hash functions.

Recently, the work by Sehat et al. [10] bypasses the programmable random oracle challenge and introduces a unique privacy-aware deduplication mechanism for multi-client environments called dual deduplication. In dual deduplication, clients perform information-theoretic transformations on their data before uploading it to the CSP. These transformations aim...
at puncturing the data to simultaneously increase deduplication capabilities on the CSP, and creating uncertainty for the CSP about the possible pre-images of the uploaded chunk. From a high level perspective, to the eye of a CSP or an attacker, the dual deduplication system proposed in [10] mimics a deletion channel from information theory [14]. In particular it is known that retrieving the original data after transmission through the deletion channel is a hard problem, predicting how input data will be modified by the channel is hard as well [14]. While these properties are undesirable to establish a reliable communication between two parties, dual deduplication uses them as key features for proving the privacy of the system. In this work, we build on the dual deduplication idea introduced in [10], and mitigate the three major drawbacks of the original construction. First, clients in [10] need to store a significant portion of their data locally. This is contrary to the intent of outsourcing data to a CSP and minimizing local storage. Second, the CSP in [10] must perform a brute force search to find the matches for each uploaded chunk to achieve the best compression rate. Third, the system in [10] is proven private only as long as the CSP lacks information about the probability distribution of the client’s original data. This assumption does not hold for databases that are very homogeneous, or with predictable data such as time-series data or e-mail texts.

The main goal of our paper is to face the aforementioned drawbacks and provide concrete, efficient solutions to address these issues. To this end, we propose Bonsai, a fully fledged, yet scaled down (in terms of storage requirements and complexity) alternative to the Yggdrasil approach proposed in [10]. The name Bonsai is chosen to maintain a connection to and provide figurative comparison with the original dual deduplication method Yggdrasil, named after the tree of life in Norse mythology. Moreover, in Bonsai the CSP transforms and stores the received data in a tree structure that resembles a bonsai, which is a small-size version of a tree.

A. Our Contribution

We propose a new transformation mechanism for dual deduplication that improves the scrambling of outsourced data, and achieves better privacy guarantees than [10]. Compared to [10], our method requires less storage on the client side and maintains good compression rates on the CSP. This is done by employing a Pseudo-Random Number Generator (PRNG) to determine the positions of the symbols to be deleted. This change removes the need to store the positions of deleted elements on the client side, thus effectively reducing the storage requirements from 18% in [10], to only 5% in Bonsai.

We present an innovative deduplication mechanism that lets the CSP identify possible deduplications efficiently (instead of bruteforcing as in [10]). Our technique stores bases on the CSP in a forest data structure, which reduces the computational cost of finding duplicates and adding new bases as well as reducing the storage cost for the bases. In addition, we improve the compression potential on the CSP using an approach inspired by Huffman Codes and replacing the symbols with lower probability distribution. Our method achieves a competitive compression rate of 68% in the cloud, while allowing deduplication and maintaining privacy guarantees.

We prove our proposed method to be privacy-aware, in the sense that an honest-but-curious CSP faces a high degree of uncertainty when guessing what the original data of the clients is. In contrast to [10], this claim holds even assuming the CSP has knowledge of the distribution of clients’ original data, as well as the number of performed deletions.

Finally, to showcase the performance impact of Bonsai on a cloud storage system, we used a DVI dataset of 10 GB and collected simulations for compression ratios achieved by Bonsai, Yggdrasil [10], data deduplication (DD) [14], and Brotil [3]. The comparison is illustrated in Fig. 1, where plots display the ratios between the total storage required after compression (considering both client and CSP) and the size of the actual data. A compression rate of 1 indicates no compression and values above 1 expand the size of the data. The lower the compression rate, the less storage is needed for each original bit (i.e., better compression). As Fig 1 suggests, Bonsai has better compression rate than its predecessor Yggdrasil [10], especially on larger chunk sizes. Brotil outperforms Bonsai in compression rate, which is expected as Bonsai is mostly a deduplication technique, which tend to have lower compression rates than universal compressors. However, Brotil provides no privacy guarantees, in contrast to Bonsai. Interestingly, combining Bonsai with Brotil (i.e., using Brotil to compress the result of the deduplication process) yields much better compression than each individual approach. This suggests that Bonsai creates highly compressible data, even when the original data is homogenous. Finally, applying data deduplication to find repeated blocks of data before applying Bonsai and Brotil further increases the compression gain for low chunk size. Intuitively, this is due to the fact that Bonsai adds metadata to find matches across different data blocks, which is unnecessary for deduplicating identical chunks. The gains are seen for smaller chunk sizes, disappearing for larger chunk sizes due to the fact that the probability of finding an exact duplicates is reduced as the size of the chunks increases.

The outline of this paper is as follows. In Section II we describe the related work on (generalised) deduplication and
privacy in systems using deduplication. In Section III we present our system model. In Section IV we propose Bonsai and clearly describe its core algorithms and processing steps. Section V includes the analysis of the system in terms of privacy, compression rate and computation cost. Section VI includes our simulation results and discussion about the performance of Bonsai. Finally, we conclude in Section VII.

II. RELATED WORK

Data deduplication is a powerful way to reduce the storage footprint of highly correlated data [8, 16]. Delta deduplication falls into the category of redundant data reduction techniques, which also include Delta Compression techniques such as Xdelta [17]. The big difference between the two methods is the fact that Delta encoding techniques use copy/insert instructions to record the “difference” between two files on string-level, while the data Deduplication techniques remove the “redundancy” between files, either on file-level [13] or chunk-level [19]. The theoretical bounds of deduplication and generalized deduplication are discussed extensively in [9], [20], [21]. Implementations of generalized deduplication have been proposed for file systems [22] and CSPs [23].

The introduction of deduplication in the early 2000s sparked many research directions aiming at improving or enhancing the compression method. Problems and proposed solutions can be summarized into three major categories.

- **Increasing performance and throughput:** This line of research includes, but is not limited to optimization of deduplication [15], acceleration of computational tasks [24], reduction of the access time to metadata [25], and improvements of the chunking procedure based on the content [26], [27].

- **Reducing the storage requirement:** These solutions focus on reducing the fingerprint of the data. Some of the major research outputs in this category deal with reducing the overhead of the metadata [28], improving the efficiency of deduplication in the cloud and reducing the size of fingerprints [29], and reducing the overhead of the data by establishing a communication between the cloud and the clients [30]–[32].

- **Improving the privacy and security of the clients’ data:** Privacy issues arise in Data Deduplication as all the data from the clients are deduplicated together, leaving the storage system vulnerable to leaking the information of the data. As this category is the closest line of research to the solution proposed in this paper, we take a closer look at this category.

For privacy-aware data owners (clients), the CSP should be able to provide storage solutions without needing access to the content of their data. If conventional encryption is used to protect clients’ data, the compression potential on the cloud severely worsens [12]. As an alternative, Message Leaked Encryption (MLE) [33] has been proposed, which allows one to encrypt data securely, while preserving the deduplication potential. The most widely used MLE technique in deduplication techniques is called Convergence Encryption (CE) [34]. However, CE has been shown to be exploitable by brute-force [35], and is vulnerable to frequency attacks [36]. The proposed adjustments to improve CE, such as DUPLESS [37], and Duan’s scheme [38], require a trusted cloud or a third-party dealer to ensure the privacy of the data. Alternative approaches rely on the Blockchain technology [39], on gateways [40], sacrifice computational efficiency [41], or generate large overhead for the clients [42]. More recently, Multi-Key Revealing Encryption [13] has been proposed to ensure privacy of the clients’ data while maintaining deduplication capabilities. This solution uses programmable random oracles which are not instantiable in practice.

There have been surveys that describe the privacy solutions in more depth [43], [44]. We also note that [9] provides a general survey on deduplication with a comprehensive study of the state-of-the-art.

III. SYSTEM MODEL AND PERFORMANCE METRICS

In this section, we describe our system model, the type of data handled by the system, the attacker model and, finally, the performance metrics used to evaluate Bonsai.

A. System Model

We adopt the dual deduplication system introduced in [10]. Concretely, our system $S$ consists of a number of independent users, referred to as Clients, and a single CSP, referred to as Cloud. The Clients’ goal is to maximize the data they outsource to the Cloud while retaining some level of privacy on each record; the Cloud’s goal is to minimize the space required to store the Clients’ outsourced data. The system $S$ allows the Clients to locally process their data, by applying a transformation $\text{trf}$.Client on their data before upload and storing the information about the transformations locally. Note that the Cloud has no access to the information stored locally by any Client. This fact is crucial to argue the privacy of Bonsai. Finally, we assume that the communication channel between the Clients and Cloud is authenticated and error-free.

B. Data Type

The data handled by Cloud and Clients in $S$ are files $F$ represented as strings of $n_F$ symbols, where each symbol is a value consisting of $k$ bits. i.e., our alphabet is $\mathbb{A} = \{0, 1\}^k$ and files are $F \in \mathbb{A}^{n_F} = \{0, 1\}^{kn_F}$. We denote the size of the alphabet as $N = |\mathbb{A}| = 2^k - 1$.

Recall that each Client performs some transformations $\text{trf}.\text{Client}$ on their files $F$ prior to upload. In what follows, we denote the output of $\text{trf}.\text{Client}$ as an outsource-local pair $(F', L) \leftarrow \text{trf}.\text{Client}(F)$, where $F'$, called outsource is the data outsourced to the Cloud; and $L$ is the secret piece of information stored locally by Client. We call this secret information client-side deviation. Note that both outsource and client-side deviation are strictly shorter than the original file; namely the length of outsource $|F'| = n_{F'} < n_F$ and the length of the client-side deviation $|L| < n_F$. 
C. Attacker Model

Our adversary $A$ is a honest-but-curious Cloud. Concretely, $A$ has access to the whole data stored in the Cloud, i.e., any record $F'$ ever outsourced by any Client. The goal of the adversary is to breach Client’s privacy by retrieving their original data $F$, i.e., to identify the complete string $F$ from which the Client derived the outsource $F'$. We recall that $A$ has no access to the client-side deviation $L$ stored by the Client locally. In this work we consider two types of adversaries:

- A weak adversary, $A_{\text{weak}}$, does not have any information on the distribution of clients’ original data $F$. This weak adversary simulates settings where clients upload unpredictable data, e.g., very diverse types of files, or uncorrelated information.

- A strong adversary, $A_{\text{strong}}$, knows the distribution $D$ of the original files, i.e., the probability distribution with which a specific string $F$ is produced by a Client. Notably, $A_{\text{strong}}$ has knowledge of the correlation between the symbols in $F$.

As expected, the strong adversary is more realistic than the weak one. While it is reasonable to assume that in real systems the CSP may have some information about the files generated by the clients, knowing the exact probability distribution for any possible file $F$ is a requirement hard to meet. Hence, we believe a real-world attacker lies somewhere in-between the two types of adversaries we consider.

Our protocol, Bonsai, makes minimal use of cryptographic tools. Namely, the only cryptographic primitive required is a PRNG (Pseudo-Random Number Generator) function, run by the Clients within the trf.Client process. All other transformations employed in Bonsai are information theoretic. To further strengthen our security outcomes, for each type of adversary, we consider two potential scenarios:

- Case 1: $A$ has negligible probability of breaking the PRNG (this correspond to a classical adversary that is computationally bounded and runs in probabilistic polynomial time).

- Case 2: $A$ can break the PRNG (this corresponds to quantum adversary that is computationally unbounded and equipped with a quantum computer, or an adversary that knows the seed input to the PRNG).

We remark that, in Bonsai the PRNG is employed to provide better compression on the Client side. Therefore, it is possible to obtain a complete information theoretic secure version of protocol by replacing the use of the PRNG with a truly random function, though this impacts negatively the performance of the system by reducing the compression potential on the Cloud.

Finally we argue that assuming $A$ is not fully malicious is needed to guarantee the correctness of the overall system. For example, if the Cloud tampers with outsourced items, Clients may reconstruct incorrect data and the service loses reliability. We consider that in the specific setting of outsourced storage, an honest-but-curious attacker is more realistic than a malicious one. For this reason, we leave the analysis of the implications of having a malicious Cloud or a malicious Client as future work.

D. Compression Ratios

The main goal of Bonsai is to reduce the fingerprint of the data via dual deduplication (i.e., compressing data on both the Client and the Cloud sides). Therefore, it is natural to define performance metrics that measure the compression ratio of the system by looking at the Client, the Cloud, and the overall system perspective. We employ the performance metrics introduced by Yggdrasil, the state-of-the-art work on dual deduplication \cite{10}.

In what follows, $DB$ denotes the set of all the data of all the Clients; and $DB_i$ denotes the set of the data of a single Client $i$. Both $DB$ and $DB_i$ contain a number of files $F$. We assume that our system $S$ has a total of $n$ Clients, which are enumerated in a set $U = \{\text{Client}_1, \ldots, \text{Client}_n\}$. In order to formally define the compression rate of Bonsai in different parties (Client and Cloud), we use a function $\text{size}(\text{party}, D)$, which we define as follows.

$\text{size}(\text{party}, D)$: On the input of $\text{party} \in \{\text{Client}, \text{Cloud}\}$ and some data $D$, this function outputs the total number of bits required to store $D$ in the given party.

Equipped with this notation we can define the Client Compression ratio as:

$$c_{\text{Client}} = \frac{\text{size}(\text{Client}_i, DB_i)}{|DB_i|},$$

where $|DB_i|$ denotes the total size of the data in the possession of Client$_i$.

As the Cloud stores data from all the clients, we calculate the total compression ratio of the Cloud with respect to $DB$, therefore, the Cloud compression rate is defined as:

$$c_{\text{Cloud}} = \frac{\text{size}(\text{Cloud}, DB)}{|DB|}.$$  

Another interesting compression metric is the total compression rate, which indicates the total storage required by all the parties (Clients and the Cloud) to store the whole database $DB$ compared to the size of the original $DB$. This metric allows us to compare the performance of Bonsai with other compression methods that only store the data on the Cloud. Intuitively, the total required storage in Bonsai is equal to the summation of the required storage in all Clients and the required storage on the Cloud, therefore, the total compression rate is:

$$c = \frac{\text{size}(\text{Cloud}, DB) + \sum_{i=1}^{n} \text{size}(\text{Client}_i, DB_i)}{|DB|}.$$  

E. Privacy Metrics

We estimate the privacy level of a dual deduplication system $S$ in terms of the uncertainty metric $\mathcal{U}$ and the leakage $L$. The uncertainty metric shows the amount of unpredictability that an adversary $A$ faces when it attempts to guess the value of a given record $F$, having seen only its outsourced version $F'$. We acknowledge that the adversary may have some knowledge about the file $F$ prior to receiving the $F'$, therefore, after receiving the $F'$, the uncertainty of the adversary drops as it gains more information about the file $F$. The leakage $L$
measures this drop in the uncertainty of the adversary $A$ after receiving $F'$, i.e., the amount of information about $F$ leaked to $A$ after receiving $F'$. We use the notation of Shannon’s entropy [45] to calculate the uncertainty metric and leakage for our adversaries. We assume that $A$ uses its knowledge to generate the list of all possible preimages $I \leftarrow A(D, F')$ of a given base $F'$. It is clear that the original file belongs in this list, $F \in I$. Then the adversary tries to predict the real value of $F$ using the preimages in $I$ and the probability that each one of them is the real $F$. The uncertainty that the attacker faces to guess the original file $F$ is equal to the entropy of the set $I$. Therefore, we define uncertainty metric as follows.

**Uncertainty Metric:**

$$U(F) = H(F|F'),$$

where $H(F|F')$ is the Shannon’s entropy of $F$ given $F'$. Based on our definition, the uncertainty prior to receiving $F'$ is equal to $H(F)$. Therefore, the leakage of information that occurs when sending $F'$ is calculated as follows.

**Leakage:**

$$L = \frac{H(F) - H(F|F')}{H(F)}.$$  

Note that the Shannon’s entropy of file, i.e., $H(F)$ is calculated in the viewpoint of the adversary. This means that if we consider the weak adversary, the value for $H(F)$ is equal to the Shannon’s entropy for a uniformly random probability distribution and not the actual value of the Shannon’s entropy based on the statistical properties of the database $DB$.

### IV. THE BONSAI PROTOCOL

We begin by providing an overview of Bonsai followed by a detailed description of the actions performed in the Cloud and the Client. We conclude with a detailed explanation of the whole the protocol.

#### A. Bonsai in a nutshell

We employ a similar system model as [10], namely, a single Cloud storage provider connected to multiple Clients. Clients wish to outsource their data to the Cloud in a privacy-preserving manner, while the Cloud wishes to reduce the storage footprint of the received outsourced data.

To achieve privacy-preserving dual deduplication, each Client applies a transformation $\text{trf.Client}(F)$ to the original file $F$ to generate the pair $(F', L) \leftarrow \text{trf.Client}(F)$. Intuitively, $F'$ is the record Client outsources to Cloud, and this should leak as little information about the original file $F$ as possible (to preserve privacy). $L$ is called the client-side deviation and condenses instructions on how to reconstruct $F$ from $F'$. The $\text{trf.Client}$ entails a number of manipulations as shown in Section [IV.C] and Fig. 3. For the sake of simplicity, we describe only the main manipulation in $\text{trf.Client}$, i.e., Del (deletion), in more detail. As the name suggests, this transformation removes symbols from $F$. The outcome is a sub-string $F'$ of the original file $F$. To build intuition, Del acts as a deletion channel. Thus, it simultaneously performs two tasks (1) it reduces the size of the outsourced record, and (2) it makes it harder for the Cloud (and thus an adversary $A$) to guess the exact original data $F$ from $F'$. To enable the reconstruction of $F$ from $F'$, Del returns an additional output that contains the list of deleted positions (in the form of pointers) and the corresponding deleted values (in the form of alphabet symbols) in form of the client-side deviation $L$. With $L$ at hand, the Client can efficiently reconstruct $F$ from $F'$. Fig. 2 illustrates an example of how Del works. In this example, the symbol ‘1’ from position ‘6’ is deleted. The local deviation contains two values: ‘6’ is a pointer indicating the position of the deleted symbol, and ‘1’ is the deleted symbol.

In Bonsai, we employ a PRNG function to select the positions where we perform deletions. Concretely, the Client selects a pre-determined number of seeds for each chunk, and interprets the output of the PRNG of each seed as a sequence of positions on which to apply Del in the chunk. The use of seeds instead of storing positions alongside values, as done in [10], significantly reduces the storage requirement on the Client. This means that all positions of deleted elements in $L$ can be stored by a single seed. Further details on how to combine the PRNG and Del, and the other manipulations involved in $\text{trf.Client}$, are given in Section [IV.C]. We also analyze how the use of a PRNG impacts security in Section [V.E].

Bonsai employs a novel way to perform generalized deduplication on the Cloud side. Upon receiving an outsource $F'$, instead of looking for similar bases according to classical distance metrics, e.g., Hamming [9] and Swap [10], the Cloud in Bonsai processes $F'$ to generate a sorted base $B$ and a set of strings, called cloud-side deviation. After this transformation, the Cloud applies deduplication on $B$. We call this Cloud-side process $\text{trf.Cloud}$. Its aim is to identify an alternative representation of $F'$ that deduplicates with higher probability.

The manipulations in $\text{trf.Cloud}$ are: grouping, Huffman coding and sorting as detailed in Section [IV.D] To give an intuition, the Cloud in Bonsai identifies each symbol with three values. We denote these values by $\text{BracketID}$, $\text{SymbolID}$, and $\text{ValueID}$. These three values uniquely identify each symbol (more details in section [IV.D]). Upon receiving an $F'$, the Cloud generates a string containing the $\text{BracketID}$ of all symbols in $F'$, and sorts it in ascending order. This string is called base and is denoted by $B$. As the base $B$ is sorted, the Cloud stores the $B$ of all of the received $F'$ in a forest structure (called $B$), to help find duplicates in time $O(\log |B|)$, where $|B|$ is the number of bases in $B$. Using this forest structure, instead of directly storing the $B$ for each $F'$, the Cloud assigns a pointer to $B$ in $B$, and uses this pointer to store each $F'$. The Cloud needs to store the necessary information to reconstruct $F'$ from $B$ on Client’s request. This information is stored in what we call cloud-side deviation. Cloud-side deviation contains of three strings: $S$, $A$, $C$. In detail, $S$ contains information about the position of the symbols in the original outsource $F'$; $A$ contains the $\text{SymbolID}$ of the symbols; and $C$ contains the $\text{ValueID}$ of the symbols. $S$ is created by using an algorithm inspired by Cycle Sort [46] to
identify the required swaps needed to sort $B$. This algorithm is detailed in Algorithm 1 in Section IV-D. To order to reduce the size of the SymbolID and ValueID for the symbols, and ultimately having a bigger compression gain in the Cloud, the Cloud uses Huffman coding to represent the SymbolID and ValueID of the symbols in each Bracket. We discuss the choice of Huffman coding and its implications in Section IV-D.

B. Policy and protocols between Cloud and Client

The communication between Cloud and Client consists of two algorithms: Upload and Get. With Upload the Client sends $F'$ to the Cloud; while with Get the Client requests back the uploaded data from the Cloud. In order to ensure consistency, the Client uses a file identifier, fid, for each outsource $F'$ it uploads to Cloud. The identifier fid is sent to the Cloud alongside the $F'$ and is used by Get as a reference to $F'$. Since we assume Cloud not to be malicious, fid does not serve as an integrity tool. Employing a cryptographic secure hash function would, however, mitigate some attack scenarios in the presence of a malicious cloud, or provide secure file sharing among Clients [17].

In Bonsai, the Cloud performs deduplication to optimize space and reduce the storage requirements. Therefore, the compression potential increases if the received outsource $F'$ are similar to each other. This is achieved by enforcing a certain probability distribution on the uploaded records. In order to ensure that the Cloud receives deduplication friendly data, Bonsai adopts policies on the probability distribution of symbols in $F'$. Policies are publicly available to Clients and should be fetched before Upload and taken into consideration when running trf.Client. In detail, Policy $= [D', n_{F'}]$ where $n_{F'}$ is the expected size of $F'$ and $D'$ is defined as an array of values $D' = [p_0', \ldots, p_{N-1}]$, and $p_i'$ indicates the probability of symbol $i$ in $F'$. This policy is calculated by the Cloud from the set of stored bases $B$. This increases the potential for deduplication between outsourced data by creating $F'$ that are similar to each other, leading to higher deduplication rates and lower storage requirement in the Cloud and ultimately higher compression gain. The Policy may include more detailed information to further increase compression capability of Cloud or for other use cases such as ensuring reliability. We leave other information that can be included in Policy and the effect of such information on performance and compression potential of Bonsai to future work.

C. The Client of Bonsai

The Client that wishes to upload its data $F$ to the Cloud, receives a policy $\text{Policy} = [D', n_{F'}]$ that is generated by the Cloud based on the already stored data. Following the Policy, the Client performs Del until the string $F'$ has size of $n_{F'}$. In Bonsai, deletions are carried out on locations identified by the output of a seeded Pseudo-Random Number Generator (PRNG). More specifically, the Client does not need to store the positions of the deleted symbols. The seed of the PRNG and the deleted values suffice to reconstruct the original file $F$ from $F'$. To make sure the output of trf.Client fits the Policy in the best possible way, for each file $F$, the Client generates a set of $t$ PRNG seeds, where $t$ is a Client-wide constant, and unique for each Client. If the value of $t$ is large, there is a higher opportunity for the the generated $F'$ to follow the probability distribution indicated in the Policy. However, the Client has to allocate more time and computational power to perform the transformation for each seed.

Fig. 3 illustrates a complete example of trf.Client, i.e., the transformations the Client performs on $F$ prior to Upload. In this example, we consider the same $F$ as in Fig. 2 $k = 4$, and $t = 2$, and we assume that the policy Policy is:

\[
\text{Policy} = \frac{3}{16} \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 3 \\
16 & 1 & 1 & 1 & 1 & 3 & 3 & 3 & 3 & 3 & 3 & 8 & 8 \\
16 & 24 & 24 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64
\end{bmatrix}
\]

(6)

In this case, the Client starts by generating two random seeds, namely $\text{seed}_1$ and $\text{seed}_2$. By using $\text{seed}_1$ and $\text{seed}_2$ as the seeds the PRNG, the Client generates two sets of values, each containing $n_{F'} - n_{F''} = 3$ elements. The generated set of values indicate the position of the symbols to be deleted for the construction of the respective outsource strings $F'_1$ and $F'_2$. This step is shown as step 1 in Fig. 3. In this particular example, $\text{seed}_1$ generates the set of positions $\{5, 8, 2\}$ and $\text{seed}_2$ generates the set $\{6, 1, 3\}$.

In step 2 of Fig. 3 the Client proceeds to delete the symbols in the given positions, storing the values of the deleted symbols in the order of deletion in $L$. The result is a potential outsource $F''$ and a client-side deviation $L$. At this point, $L$ contains the random seed that was used to generated the $F''$ and deleted values in the order that the PRNG outputs the positions. In this particular example, considering seed $s_1$ generates 5, 8, and 2 in that order, the first values stored in $L$ is the value in position 5, i.e., 7, followed by the value in position 8, which is 2 and finally the value in position 2, i.e., 1. In order to increase the possibility of generating a string that best adheres to the probability distribution $D'$ in Policy, we make use of another well-known information-theoretic transformation: invert (Inv). In the binary case, when a string is transformed by Inv, all bits that has the value of 1 are flipped to 0 and vice-versa. More generally, Inv replaces each symbol $i$ with its complement $N - i - 1$. In step 2 of Fig. 3 we show how the inverted
Hamming distance, Euclidean distance, or Manhattan distance. In this work, the notion of closeness between the probability distribution $D'$ and $D_{F'}$ is calculated by the Euclidean distance between the two vectors, i.e., the selected $F'$ is the one with minimum value of $l_{F'} = \sqrt{\sum_{i=1}^{N} (p_i^{F'} - p_i^{F})^2}$. Hence, as shown in Step 3 in Fig. 3, the Client first calculates the frequency of symbols in each $F'$, creating $D_{F'}$. Fig. 3 shows the calculated $D_{F_4'}$ for $F_4'$. We see that $F_4'$ has one element 0, therefore, $\#0 = 1$ and $p_0^{F_4'} = 1/8$. After generating the distribution of symbols for all generated $F'$, the Client calculates the value of $l_{F'}$. This calculation is carried out for $F_4'$ in Fig. 3. After calculating the value of $l_{F'}$ for all generated $F'$, Step 2, the Client chooses the $F'$ with the minimum value of $l_{F'}$ as the last step (Step 4). In the given example, $F_4'$ has the lowest value of $l_{F'}$, and therefore is chosen by the Client to be outsourced to the cloud. Then, the client proceeds with generating fid for the selected $F'$, sending fid and $F'$ to the Cloud, while storing fid and $L$ locally.

### D. The Cloud of Bonsai

As mentioned, the Cloud uses an innovative method to store the received $F'$ and find possible deduplications in the received data. In this section, we explore the ideas and theory that lead to trf.Cloud, and describe trf.Cloud and deduplication in the
Cloud in more detail. We also show a toy example to illustrate trf.Cloud visually.

a) Motivation: The main idea behind trf.Cloud is to reduce the time needed to find potential deduplication between received data in the Cloud. The idea of trf.Cloud consists of four main ideas to increase the performance and compression rate in the Cloud. These ideas include splitting data in brackets, sorting, Huffman coding and changing values.

Data Split using Brackets. This process organizes symbols in brackets. Each bracket is assigned a BracketID and each symbol in the bracket is identified with a SymbolID. The brackets are pre-defined and unique for each value of \( k \). Table I gives an example of the bracket system where \( k = 4 \), where each column represents the BracketID and each row represents the SymbolID of a given symbol. Using this table, the Cloud transforms the received outsource \( F' \) into two different strings: 1. A base \( B \) which includes the BracketID of the symbols, sorted in ascending order, which is used for deduplication; and 2. An addendum \( A \) which includes the SymbolID of the symbols, which is added to the cloud-side deviation. In order to reduce the number of bracket IDs used we divide the aforementioned table into four zones. Each zone is a sub-table of size \( 2^\frac{k}{2} \). At its core, this approach is a sub-categorization where rarely used BracketID values are omitted and the symbols are fitted into fewer Brackets. This is detailed in the Change Values section.

Sorting. After performing the data split using brackets, the generated \( B \) is sorted. A String \( S \) indicating the set of swaps performed to sort the \( B \) is stored in cloud-side deviation. The \( B \) is stored in a forest data structure to ease the search process and further compress the bases. In order to reverse the process of sorting, we store the required swaps (Swap) needed to sort \( B \). Due to the split in brackets, the Cloud can perform swaps for \( 2^\frac{k}{2} \) values instead of \( N = 2^k \) values, reducing the overhead and time complexity to store and find the required swaps.

Huffman Coding. We leverage the fact that the \( F' \) received by the Cloud follows a certain probability distribution (ensured by the policy), and use Huffman coding in order to reduce the expected size of the SymbolID that the Cloud needs to store in the cloud-side deviation. To generate the SymbolID of the symbols based on their probability distribution, the Cloud creates a table with \( \frac{k}{2} \) columns, which represent the BracketID of symbols. Then, the Cloud populates the table by putting the symbols with highest probability distribution in the table first.

The way that the table is populated depends on the particular structure of the data. In this work, we populate the table by filling the rows first. An example is illustrated in Fig. 4 for \( k = 4 \). Other methods for filling the table are possible, e.g., columns first, with implications in compression and system performance. By filling the rows first, the Cloud optimizes the size of SymbolID for the symbols, as the symbols with higher probability are assigned the same SymbolID. However, as the frequent symbols are assigned to different BracketID, the average number of swaps would be high. On the other hand, if the table is filled by columns first, the expected size of SymbolID is higher, but the average number of swaps is smaller. The choice of the method and the optimal outcome for the compression rate heavily depends on the structure and statistical characteristics of the files in \( DB \).

After populating the table, the Cloud treats each row as a variable, and calculates the probability of each row, which is equal to the summation of the probability of the symbols in the given row. Then, the Cloud generates the Huffman code for the rows of the table, and sets the value of the Huffman code as the SymbolID of that row. As an intuition, consider the same probability distribution as the example in Eq. (6). In this case, the procedure of generating the BracketID and SymbolID of symbols is illustrated in Fig. 4.

### Change Values

The SymbolID for symbols with high probability distribution is significantly shorter than the symbols with lower probability. However, the size of SymbolID for symbols with lower probability distribution is large, and there-

---

**TABLE I**

| Probability | Symbol |
|-------------|--------|
| 3/16        | 0000   |
| 1/8         | 1111   |
| :           | :      |
| 3/64        | 1101   |
| :           | :      |
| 1/24        | 0011   |

| BracketID   | Sum of probabilities of each row |
|-------------|---------------------------------|
| 00 01 10 11 | 3/16+1/8+1/16+1/16 = 7/16 |
|            | 1/16+1/16+3/64+3/64 = 7/32 |
|            | 3/64+3/64+1/24+1/24 = 34/192 |
|            | 1/24+1/24+1/24+1/24 = 1/6 |

| SymbolID |
|----------|
| 0        |
| 1        |
| 0        |
| 110      |
| 111      |

**Fig. 4.** An illustration of generating Brackets using Huffman codes for \( k = 4 \) (\( A = \{0, 1\}^4 \)) and \( D \) as in Eq. (6)
fore, these symbols have diminishing effect on the compression potential. It is also beneficial to have as few BracketID as possible to reduce the number of swaps in the sorting stage. Thus, we propose using Change Value transformation (ChngV) and describe it for $k = 8$. Using this transformation, the Cloud changes the value of symbols with the lowest probability to symbols with the highest probability. In order to perform this transformation, we divide the brackets table into 4 zones, as described in the Bracket stage. Table II shows an example for $k = 8$. In this table, we assume that the symbols are sorted based on their probability distribution, where $p_0$ denotes the symbol with the highest probability, followed by $p_1$, and so on, e.g., $p_{255}$ is the symbol with the lowest probability. The Cloud fills the table by filling the first zone, which is the zone on the top left corner, followed by the second zone on the top right corner and then bottom left and bottom right. In each zone, the symbols are put in their place using the rows first methods, as discussed in the Huffman coding section. Then, the Cloud assigns a a variable for each row of all four zones, generating the SymbolID for each row as discussed earlier. Also, each zone is assigned a Value ID ValueID. The ValueID for each zone is generated using the same idea of SymbolID.

In the end of this procedure, each symbol has a position $[i, j]$ in its zone, an assigned BracketID, SymbolID and ValueID. Upon receiving an outsource $F'$, the Cloud first performs a change value operation. For each symbol, the Cloud finds the position of the symbol in its zone in the table, Assuming the symbol is in position $[i, j]$ of one of the zones, the Cloud change the value of that symbol to the value in the position $[i, j]$ in the first zone (top left corner). Then, the Cloud stores the ValueID for the symbols in a string $C$, which is added to the cloud-side deviation. The transformed string is stored and is used to create $B$ and $A$ as discussed before. Using this technique, the data becomes more homogenous, reducing the potential number of BracketID, and therefore reducing the expected number of swaps, while also reducing the expected length of the SymbolID for the symbols by eliminating the symbols with lower probability distribution. This improvement is achieved by only storing an overhead value of ValueID for each symbol, which is smaller for symbols with higher probability distribution.

| TABLE II |
| --- |
| AN ILLUSTRATION OF BRACKETS USED FOR tfr.Cloud FOR $k = 8$. |

![](image)

**Algorithm 1** The algorithm to find swaps

```plaintext```
Initiate `swaps` as an empty vector.

\[ j \leftarrow 0 \]

while \( j < n_F \) do

\[ \text{if } B[j] \neq H[j] \text{ then} \]

Find minimum \( k \) where \( B[k] = H[j] \neq H[k] \)

Swap \( H[j] \) and \( H[k] \)

Add \((j, k)\) to `swaps`

end if

\[ j \leftarrow j + 1 \]

end while
```

\[ b) \text{tfr.Cloud: Let us describe in detail the journey on an } F' \text{ as it reaches the the Cloud. The Cloud has already generated the Brackets table before receiving } F'. \text{ We support our description using an example for } k = 4, \text{ where the Bracket table is given in Table III. Fig.5 illustrates the actions performed by the Cloud after receiving an outsource. For the sake of simplicity, we assume that in our example, the ValueID for the zones are 00, 01, 10, 11.} \]

In the first step, if \( k = 8 \), the Cloud changes the value (ChngV) of the symbols to the respective symbol in the first zone of the brackets table, while storing the value id ValueID of all symbols in a separate string denoted by $C$. This step transforms the received $F'$ to a new string denoted by $G$, which includes the values of the symbols after ChngV (only 4 possible values). This step is illustrated as Step 1 in Fig.5.

As an example, the value of symbol “10” in position 1 is going to change. Looking at the Brackets table, the binary representation of “10”, i.e., “1010” is in position $[0, 0]$ of zone 4, therefore, its value is changed to the symbol in position $[0, 0]$ of zone 1, which is “0000” in Binary, i.e., “0”. The Cloud inserts the ValueID of zone 4, i.e., “3” into $C$.

After generating $G$, the Cloud uses the Brackets table to separate each symbol into its BracketID and SymbolID, creating two strings. 1) String $H$ which contains the BracketID of the symbols, and 2) String $A$ which contains the SymbolID of the symbols. This step is illustrated as step 2 in Fig.5. As an example, the first symbol in $F_4'$, i.e., 4 has a BracketID = 1 and a SymbolID = 0. Therefore, the Cloud inserts 1 in the first position of $B$, and 0 in the first position of $A$.

In the next step, the cloud sorts $H$ to generate $B$. We use Mergesort [48] as a quick and scalable sorting algorithm in order to generate $B$ from $H$. In order to reconstruct $H$ and ultimately $F'$ later in decompression phase, the Cloud stores

| TABLE III |
| --- |
| THE BRACKETS TABLE FOR THE TOY EXAMPLE OF FIG.5 |

| BracketID |
| --- |
| 0 | 1 | 0 | 1 |

| SymbolID |
| --- |
| 0 | 0000 | 0001 | 1000 | 1100 |
| 1 | 0010 | 0101 | 1001 | 1110 |
| 0 | 0011 | 0111 | 1011 | 1111 |
### Step 1: ChangeValues

| BracketID | 0 | 1 | 0 | 1 |
|-----------|---|---|---|---|
| SymbolID  | 0 | 0000| 0100| 1000| 1100|
|           | 1 | 0000| 0101| 1001| 1101|
|           | 0 | 0010| 0110| 1010| 1110|
|           | 1 | 0111| 0111| 1111| 1111|

### Outsource \(F'_1\)

| \(G_1\) | \(C_1\) |
|---------|---------|
| 4 10 8 9 1 2 12 15 | 0 1 4 5 1 1 0 1 |

### Step 2: Separation

| BracketID | 0 | 1 | 0 | 1 |
|-----------|---|---|---|---|
| SymbolID  | 0 | 0000| 0100| 1000| 1100|
|           | 1 | 0000| 0101| 1001| 1101|
|           | 0 | 0010| 0110| 1010| 1110|
|           | 1 | 0111| 0111| 1111| 1111|

| \(G_1\) | \(A_1\) |
|---------|---------|
| 1 0 0 1 1 0 4 5 | 1 0 0 1 1 0 0 1 |

| \(G_2\) | \(A_2\) |
|---------|---------|
| 0 1 1 4 5 1 1 0 1 | 0 1 0 1 1 1 1 0 1 |

### Step 3: BaseSort

| \(H_1\) | \(B_1\) | \(S_1\) |
|---------|---------|---------|
| 1 0 0 0 0 0 1 1 | 0 0 0 0 0 1 1 1 | 1 0 0 0 0 0 0 0 |

| \(H_2\) | \(B_2\) | \(S_2\) |
|---------|---------|---------|
| 0 0 1 1 0 0 0 0 | 0 0 0 0 0 1 1 1 | 0 0 1 1 0 0 0 0 |

### Step 4: Deduplication

\(0 - 0 - 0 - 0 - 0 - 0 = 1 - 1 - 1 \leftarrow \mathcal{P}_1, \) (Pointer to \(B_1\))

\(0 - 1 - 1 \leftarrow \mathcal{P}_2, \) (Pointer to \(B_2\))

\(B_1\) and \(B_2\) stored in \(B.\)

---

the necessary information about the difference between \(H\) and \(B.\) This information is stored in \(S\) as the required swaps to transform \(H\) into \(B.\) In order to identify the required swaps, the Cloud compares \(B\) with \(H,\) using an algorithm inspired by Cycle Sort \[48]. Our algorithm does not sort \(H,\) instead, it attempts to find the minimum number of Swap required to transform \(H\) into \(B.\) In detail, the algorithm finds the first position \(j,\) where the symbol is different in \(H\) and \(B,\) i.e., \(B[j] \neq H[j]\) This indicates that the value needs to be swapped in order to generate \(B.\) Then, the algorithm finds the first possible position \(k > j\) in \(B\) where \(B[k] = H[j],\) so that if the symbols in positions \(j\) and \(k\) are swapped in \(H,\) the symbol in position \(j\) will be equal to the corresponding symbol in \(B.\) In order to mitigate extra swaps, the Cloud ensure that the symbol in position \(k\) is not already in correct position by checking if the symbol in position \(k\) in \(B\) is equal to the symbol in position \(k\) of \(H,\) i.e., the swap is only performed if \(B[k] \neq H[k],\) otherwise the Cloud searches for the next valid \(k.\) This algorithm continues until all the symbols are in their sorted positions. Algorithm \[1\] shows the pseudocode for the steps taken by our algorithm. Note that our algorithm is sub-optimal (finding the minimal number of swaps to make two arrays identical is NP-hard \[49\]).

After identifying the swaps, The Cloud generates \(S.\) \(S\)
consists of two parts, a bitmap of size \( n_{F'} \), and an array consisting of positions. The Cloud starts by setting all values in the bitmap to zero, and initiating an empty array. For each identified swap between two positions \( i \) and \( j \), where \( i < j \), the Cloud: 1) Sets the value of position \( i \) in the bitmap to 1. 2) Adds \( j \) to the array that stores the positions. Step 3 of Fig. 5 shows the final result of this algorithm in our toy example. In this particular example, for \( F'_1 \), only one swap is needed between the symbols in positions 0 and 5. Hence, in \( S \), the value in position 0 is set to “1”. The Cloud adds “S” to \( S \), which uniquely identifies the performed swap to be between positions “0” and “5”.

The Cloud stores all bases in a set \( B \). We leverage the fact of having sorted bases in the Cloud to create an efficient data structure to search for possible deduplications. To achieve this, we use a forest structure to store \( B \). The roots are the first elements of the stored bases and each child has the next element of its parents. Each leaf represents a single \( B \). The Cloud assigns a pointer to each leaf, which will indicate the \( B \) associated to it. Step 4 of Fig. 5 gives an intuition of how this structure is generated, given the Cloud only has the two bases of the \( F' \) shown in our example. In our case, the forest structure is composed by single tree, due to the fact that both of the bases in the Cloud have the same initial element (with value 0). After creating \( B \), the Cloud searches for \( B \) in \( B \). If \( B \) is found in the \( B \), the Cloud deduplicates \( B \) and store the pointer \( P \) to the \( B \) along with the respective \( A \), \( S \) and \( C \) of the \( F' \). Otherwise, the Cloud will add \( B \) to \( B \), assign a pointer \( P \) to it and store \( P \), \( A \), \( S \) and \( C \) to represent \( F' \). Using this structure, it is easy to see that the time complexity for searching for a \( B \) in \( B \) is \( O(n_{F'}) \) time. The addition of a new base to the tree takes \( O(n_{F'}) \) time. This is a significant improvement compared to Yegdrasil [10], where the possible deduplications are found using brute-force search, taking up to \( O(n_{F'}^2 \cdot |B|) \) time.

c) **Decompression:** When the Cloud receives a request from a Client that wants to retrieve its data, it reconstructs the \( F' \) from \( P \), \( A \), \( S \) and \( C \). In this procedure, the Cloud simply reverses the steps taken to generate the four strings.

### E. Protocol Breakdown

We describe our protocol for privacy-aware dual deduplication in multi client settings by pointing out the algorithms done in Client, Cloud and between them.

At initialization Cloud holds an initial set \( n_b \)-size strings called bases \( B = \{B_1, \ldots, B_n\} \). \( B \) may be empty at the start of the algorithm, in the case that the Cloud has not received any data yet. However, \( B \) is updated over time by the outsourced data from the Clients.

- **SetUp(\( B \)):** This algorithm is run by the Cloud periodically. It takes as input the set of bases \( B \) and outputs a policy \( \text{Policy} = \{n_{F'}, D'\} \).

- **Upload(\( \text{Policy}, F' \)):** This algorithm is run by Client. On input Policy and a file \( F \), the Client runs local transformations \( \text{trf.Client} \) to generate a base \( F' \) of size \( n_b \), and its corresponding deviation \( L \). Then, it generates a unique file identifier \( \text{fid} \). The algorithm outputs the triple (\( \text{fid}, F', L \)). The pair (\( \text{fid}, F' \)) is outsourced to the Cloud, while (\( \text{fid}, L \)) is stored locally by the Client.

**Deduplication(\( \text{fid}, F', B \)):** This algorithm is run by the Cloud. On the input a file identifier \( \text{fid} \), a string \( F' \), and the set of bases \( B \), the Cloud generates the respective \( B, A, S \) and \( C \); the Cloud checks whether \( B \in B \), in which case it stores its respective pointer, \( P \), if not, Cloud adds \( B \) to \( B \) and assigns a pointer \( P \) to it; The algorithm outputs (\( \text{fid}, P, A, S, C \)).

**Get(\( \text{fid}, L \)):** This algorithm collects the Client’s procedures of an interactive protocol with Cloud to retrieve an outsourced data item. First, Client sends \( \text{fid} \), symbolizing a request to retrieve the item that was outsourced with that \( \text{fid} \). Upon receiving a response \( F' \) from Cloud, the Client uses the information encoded in \( L \) to invert the deletions that led to \( F' \), reconstructing \( F \).

**Deco(\( DB, \text{fid} \)):** This algorithm collects the Cloud’s procedures of an interactive protocol to let a Client download an outsourced data item. Upon receiving an \( \text{fid} \) request, Cloud checks whether \( \text{fid} \) exists in the database. If not, it ignores the query. Otherwise, it retrieves the corresponding record (\( \text{fid}, P, A, S, C \)); inverts the deduplication performed by Deduplication; reconstructs the decompressed \( F' \) corresponding to the outsourced string; and returns \( F' \) back to the Client.

### V. PERFORMANCE ANALYSIS

In the following, we analyze the compression ratio achieved by Bonsai, the transformation cost in both Clients and Cloud, and discuss the privacy achieved by \( S \). Unless stated otherwise, we use \( \log(x) \) as the logarithm in base 2 of \( x \).

**A. Client Compression Ratio**

We begin our analysis with studying the compression ratio on the client side, i.e., \( C_{\text{Client}} \) in Eq. (1). For each file \( F \), the user stores the respective \( L \) and \( \text{fid} \). To compute \( C_{\text{Client}} \) for one file \( F \), we accurately measure the size of \( L \). As the Client uses a PRNG to identify the position of the deleted elements, the required size to store the positions is equal to the size of the PRNG seed. We denote this size by \( s_{\text{seed}} \). The Client performs \( \alpha = n_F - n_{F'} \) subsequent Del on \( F \), where the Cloud stores the value of all deleted symbols in \( L \). Thus, \( L \) contains a total of \( \alpha \cdot k \) bits to store the value of the deleted symbols. \( L \) also contains 1 bit, indicating whether the Inv transformation has
been applied to $F'$. Therefore, the required storage needed in Client is equal to:

\[
\text{size}(\text{Client}_i, F) = s_{\text{seed}} + \alpha \cdot k + s_{\text{fid}} + 1,
\]

where $s_{\text{fid}}$ is the size in bits of a file identifier.

If the database for the client $i$, i.e., $DB_i$, has $\#F$ files, the required storage on the Client side is:

\[
\text{size}(\text{Client}_i, DB_i) = \#F \cdot (s_{\text{seed}} + \alpha \cdot k + s_{\text{fid}} + 1).
\]

The original size required for $DB_i$ prior to the transformations of Bonsai is:

\[
|DB_i| = \#F \cdot k \cdot n_o.
\]

Thus,

\[
C_{\text{Client}} = \frac{\text{size}(\text{Client}_i, DB_i)}{|DB_i|} = \frac{s_{\text{seed}} + \alpha \cdot k + s_{\text{fid}} + 1}{k \cdot n_F}.
\]  

(8)

B. Cloud Compression Ratio

The compression rate on the Cloud, i.e., $C_{\text{Cloud}}$, according to Eq. (2), considering the different contributing effects. As described is Section IV, the cloud stores three strings $A$, $S$, and $C$ for each received $F'$, as well as the identifier $\text{fid}$ and a pointer to the base $B$. The bases are stored in $B$, where we denote the size of $B$ as the size of the forest $s_B$ used to represent all stored bases. Note that $s_B < n_{F'} \cdot |B|$, as the forest structure removes the redundancy between different bases in $B$. Thus, the total storage in the cloud is equal to the cost of storing $A$, $S$, and $C$, the $\text{fid}$ and a pointer to the respective $B$ for each received $F'$ plus the required storage for the $B$. Our experiments show that $s_B$ is very small compared to the size of the data as the Cloud is able to perform deduplication on a large number of bases.

The expected size of the $A$ is equal to the expected length of Huffman Coding, which is equal to the entropy of the data in the Cloud, i.e., $H(\text{Cloud})$. The size of $S$ is dependent on the number of swaps needed to sort the base $B$. For each Swap, the cloud needs to store 1 position for the position of the swapped elements, and each position requires $\log n_{F'}$ space. $S$ also includes a bitmap of symbols, which has the size of $n_{F'}$ bits. Therefore, the Cloud requires $n_{F'} + n_{\text{swap}} \cdot \log n_{F'}$ storage for $S$, where $n_{\text{swap}}$ is the number of swaps required to sort $H$. The number of swaps $n_{\text{swap}}$ is heavily dependent on the structure of the data and the way the files are chunked from the database $DB$. In Cloud the Cloud stores the Huffman representation of the zones, so the size of this string is also dependent on the structure and the probability distribution of the data. In worst case, the size of ValueID for each symbol is exactly 2 bits, leading to $2n_F$ required storage for $C$. In the following analysis, we use this value. In practice, Cloud uses Huffman codes to determine the size of ValueID, so the size of ValueID for most probable symbols is usually 1 bit, leading to lower required storage for use cases.

For each $F'$, we have:

\[
E(\text{size(Cloud, } F)) = H(D') + 3n_F + n_{\text{swap}} \cdot \log n_{F'} + s_{\text{fid}} + s_p,
\]

where $s_p$ is the size of the pointer or identifier to the $B$ in $B$.

Thus, the total compression rate on the cloud is

\[
C_{\text{Cloud}} = \frac{s_B + \#F(H(D') + 3n_F + n_{\text{swap}} \cdot \log n_{F'} + s_{\text{fid}} + s_p)}{\#F \cdot k \cdot n_F}.
\]  

(9)

C. Global Compression Ratio

The required storage size in Client side does not change and can be calculated as size(Client, $DB_i$) in section IV-A. Therefore, the global compression ratio of the system is given by the sum of the Client compression ratio and the Cloud one

\[
c = \frac{s_B + \#F(c + H(D') + \alpha \cdot k + 3n_F + n_{\text{swap}} \cdot \log n_{F'})}{\#F \cdot k \cdot n_F},
\]

(10)

where $c$ is a constant number, denoting the size of the pointers and the random seed of the PRNG, and is equal to $c = s_{\text{seed}} + s_p + 2s_{\text{fid}} + 1$.

D. Transformation Costs

In this section, we calculate the computation cost of the algorithms discussed in Section IV in Cloud and Client. We provide two theorems for the computational cost of operations in Cloud and Client.

**Theorem 1:** The computational cost of the procedures applied in the Cloud during Upload is $O(t \cdot n_{F'})$.

**Proof:** The Client performs Del for each file prior to upload, which is in the Upload procedure. The Client performs two set of instructions in Upload. First, performing Del for each file and PRNG seed. Second, calculating the probability distribution of the generated $F'$ to choose the one that adheres the best with the distribution $D'$ provided in the Policy.

Using a vector data structure for file $F$, results in linear complexity for deletion of $n_{F'} - n_{F''}$ elements in the size of the $F'$, i.e., $O(n_{F'})$. This can be performed by copying elements not scheduled for deletion to a different vector of size $n_{F''}$, skipping the copy of deleted elements. Since the procedure is performed $t$ times for $t$ different seeds, the total complexity of this instruction is $O(t \cdot n_{F'})$.

For each generated $F'$ and its inverted $F''$, the Client must calculate the probability distribution. Calculating the probability distribution requires reading all the symbols, i.e., $O(n_{F'})$ operations. As we have $2 \cdot t$ possible $F'$ to evaluate, this instruction has complexity $O(t \cdot n_{F'})$. Thus, the complexity at the Client is $O(t \cdot n_{F'})$.

**Theorem 2:** The computational cost of procedure in the Cloud after receiving an $F'$ form a Client during Deduplication is equal to $O(n_{F'}(\log n_{F'} + N))$ when using MergeSort for the Sorting phase.

**Proof:** The Cloud performs three sets of transformations on the received $F'$, namely ChngV, Seperation and Sort. Therefore, in order to calculate the time complexity at the Cloud after receiving an $F'$, we calculate the time complexity of these transformations.

In ChngV, the Cloud needs to (1) find the symbols in the Brackets table and (2) change the value of the symbol and insert the new value, as well as ValueID in the two output strings $G$ and $C$. The most efficient way to perform this
transformation is to search for each symbol in the outsource $F'$ instead of searching for each symbol in the table. Searching for a symbol has a time complexity of $O(n_F)$. As the Cloud has to search for $N$ symbols in total, this step has complexity $O(n_F \cdot N)$. As we use vector data structure, the Cloud can create the two output strings in linear time, i.e., $O(n_F')$. Therefore, the total time complexity of ChngV is equal to $O(n_F)$. 

During Separation, the cloud creates 2 strings $A$ and $H$ from $G$. For each symbol, the Cloud needs to find the respective value of SymbolID and BracketID from the Brackets Table. This process requires a brute-force search, and therefore has a time complexity of $O(N)$. Populating both strings is done in linear time complexity after finding the SymbolID and BracketID of each symbol. Thus, the total time complexity of this step is equal to $O(n_F \cdot N)$.

The computational cost of the transformations applied in the Cloud during Sort depends on the sorting algorithm that is used. In this work, we use Mergesort in order to sort the $H$, which has a time complexity of $O(n_F \log n_F)$. Then, the Cloud finds the swaps required using a linear comparison between the sorted base $B$ and $H$. As this algorithm is linear, it has a time complexity of $O(n_F)$. Therefore, in total, the time complexity of Sort is equal to $O(n_F \log n_F)$.

Combining these three time complexities gives the final time complexity of the Deduplication procedure.

**Theorem 3:** The computational cost of the transformations applied during the client in Get is $O(n_F)$.

**Proof:** The Client needs to insert the deleted values in $L$ into their respective positions in the $F'$. This is linear in cost with the size of $n_F$.

**Theorem 4:** The computational cost of the transformations applied during the client in Deco is $O(|B| + n_F \cdot N)$.

**Proof:** The Cloud performs Deco in four steps. First, retrieving the $B$ from $B$. Second, undoing the Swap transformations. Third, recreating the symbols from their respective BracketID and SymbolID. Lastly, restoring the changed values.

Retrieving the base from $B$ requires a search on the received fid. Although the bases are stored in a structured way, there is no guarantee that the fid is stored in the same way. Therefore, searching for a fid requires linear time dependent on the number of bases in the basisset, i.e., $O(|B|)$.

After retrieving the $B$, the Cloud needs to reverse the swaps stored in $S$ to retrieve $H$ by first identifying the required swaps. This is done by a read through $S$, that has linear complexity, i.e., $O(n_F')$. Then, the cloud performs the swaps. In the worst case, the number of swaps is linear to the size of the $B$, i.e., $O(n_F)$.

Restoring the $F'$ from $H$ requires replacing each triple of SymbolID, BracketID and ValueID with its respective symbol. Therefore, for each symbol in $H$, the Cloud needs to scan through the Brackets table to find the value associated with the triple. In the worst case scenario, the Cloud needs to scan through the whole table for each symbol, leading to a time complexity of $O(N)$ for each symbol. Therefore, in worst case, the time complexity of this action is equal to $O(n_F \cdot N)$.

Combining these three time complexities concludes the proof.

**E. Privacy Analysis**

In the following, we calculate Bonsai’s leakage $\mathcal{L}$ and uncertainty $U$ for an honest-but-curious Cloud. In detail, we consider $A_{\text{week}}$ and $A_{\text{strong}}$ as introduced in Section III.C. and for each adversary type, we investigate two scenarios. First, the adversary has negligible probability of breaking the PRNG. Second, the adversary breaks the PRNG.

1) **Weak Adversary:** Recall that $A_{\text{week}}$ has no information about the distribution of original files. This means that in its perspective, all possible values of $F$ have the same probability of being the Client’s data. Therefore, $H(F) = k \cdot n_F$. In scenario 1, the PRNG behaves like a truly random function in $A_{\text{week}}$’s view, and therefore it leaks no information about the position of the deleted elements. Given $F'$, the original string $F$ can be any of the possible strings that generate $F'$. As $A_{\text{week}}$ does not have any extra information, each potential $F$ has the same probability of being the original string, so $H(F|F') = \log(m)$, where $m$ is the number of possible pre-images of $F'$. as it has been shown in [10], we have:

$$m = \sum_{j=0}^{n_F-n_F'} \left( \frac{n_F}{j+n_F'} \right) (2^k-1)^{n_F-n_F'-j}.$$ 

Therefore, the leakage for $A_{\text{week}}$ is $\mathcal{L}(F') = \frac{k \cdot n_F - \log(m)}{k \cdot n_F}$, and the uncertainty $U$ that the $A_{\text{week}}$ faces after receiving $F'$ is $U = \log(m)$.

In scenario 2, $A_{\text{week}}$ breaks the PRNG, this means that it gains knowledge about the position of the deleted symbols. Therefore, the total number of possible pre-images of the $F'$, $m$ has a different value. However, the adversary still cannot distinguish between the different possible pre-images, as in the adversary’s viewpoint, all possible pre-images has the same probability of being the original data. In order to generate a possible pre-image, the adversary can insert a symbol in the positions that the deletions has occurred. As there are $n_F-n_F'$ positions and each position has $2^k$ possible values as the symbol, the number of pre-images of the $F'$ is equal to:

$$m = 2^{k(n_F-n_F')}.$$ 

The rest of the analysis is the same as the previous case, therefore, if the $A_{\text{week}}$ breaks the PRNG, we have the following value of leakage after receiving the $F'$:

$$\mathcal{L}(F') = \frac{k \cdot n_F - \log(m)}{k \cdot n_F} = \frac{n_F}{n_F}.$$ 

The Uncertainty for the attacker in this case is equal to:

$$U = \log(m) = k(n_F - n_F').$$
2) Strong Adversary: Recall that \( A_{\text{strong}} \) has knowledge about the probability distribution of symbols in the client. We further assume here that the adversary has a negligible probability of breaking the PRNG, i.e., has no information about the position of the deleted symbols. In this case, \( H(F) \neq k \cdot n_F \). When the cloud receives \( F' \), the original string \( F \) can be any of the possible strings that generate \( F' \). The cloud can generate all possible values of \( F \). The probability that a given \( F \) is the actual data of the Client is higher if \( 1 \cdot F \) has higher probability based on \( D \) and \( 2 \cdot F' \) can be generated by \( F \) with various deletions. We define \( W(F = F|F' = F') \) as the number of ways that \( F' \) can be generated from \( F \), i.e., number of distinct occurrences of \( F' \) in \( F \) as a subsequence. This value is calculated using a recursive algorithm and dynamic programming. Using this variable, the probability of each \( F \) in cloud’s perspective is equal to:

\[
P(F = F|F' = F') = \frac{\sum_{F} W(F = F|F' = F') P(F = F)}{\sum_{F} W(F = F|F' = F') P(F = F)}.
\]

Using the value of \( P(F = F|F' = F') \), we calculate the value for \( H(F|F') \) which is equal to the uncertainty of the \( A_{\text{strong}} \) after it receives \( F' \). The leakage of information to \( A_{\text{strong}} \) is calculated using the value of \( U \) and Eq. \( 5 \).

If the adversary breaks the PRNG, this analysis still holds, and we need to calculate the value of \( P(F = F|F' = F') \) for each possible pre-image. However, the difference between the two cases is the more information that the adversary possesses if it breaks the PRNG means that the number of possible pre-images is lower, therefore, after receiving the \( F' \), on average more information is leaked to the Cloud.

The uncertainty provided against a strong adversary is upper bounded by the uncertainty of the weak adversary, i.e., \( k(n_F - n_{F'}) \). This fact is useful to argue for privacy against a real adversary, which lies somewhere between the weak and strong adversaries. In Section \( \text{VI} \) we calculate the values of leakage and uncertainty against weak and strong adversaries and discuss the implications of it in real life scenarios.

VI. Simulation Results and Discussion

In this section, we show and discuss the performance of Bonsai in terms of its compression rate and privacy. We use three different real world datasets, including, 14 GB of email texts in enron-mail dataset \( [50] \), 18 GB of Hadoop Data File System (HDFS) logs \( [51] \) and 10 GB of DeVerse Independant (DVI) files \( [52] \). We developed a C++ implementation consisting of a Bonsai client applying deletions and a server performing deduplication algorithm described in this work. For the sake of storage friendly implementation, we defined the fid of bases to be a global variable auto-incremented by the Cloud. This is also used for the pointers to the bases in the Cloud.

Compression rate for different symbol size: Fig. \( [7] \) shows the compression rate of Bonsai for the three datasets for two different symbol size of \( k = 4 \) and \( k = 8 \). For \( k = 8 \), Bonsai achieves a total compression rate of \( C = 0.6473 \) for the DVI dataset, where the Cloud achieves a compression rate of \( C_{\text{Cloud}} = 0.6391 \), and the Client stores \( C_{\text{Client}} = 0.0082 \) of the data. For the same dataset, the total compression rate for \( k = 4 \) is equal to \( C = 0.8207 \), where the Cloud and the Client store \( C_{\text{Cloud}} = 0.8125 \) and \( C_{\text{Client}} = 0.0082 \) of the data respectively. We note that Bonsai does not drop significantly in compression potential by using diverse data, as the total compression rate for all three datasets is \( C = 0.7473 \) when \( k = 8 \) and \( C = 0.7986 \) when \( k = 4 \). This is an interesting characteristic of Bonsai which shows the potential to outperform other compression techniques in heterogenous datasets or workloads (e.g., mixed data sources with different characteristics and statistics).

Fig. \( [7] \) also suggests that for a fixed size of the original file \( k \cdot n_F = 2048 \), the total compression rate and the compression rate on the Cloud is better when the size of symbols is set as \( k = 8 \). This is because (1) Bonsai cannot reduce the size of SymbolID and ValueID using Huffman coding when \( k = 4 \), as opposed to \( k = 8 \) leading to more storage requirement for \( A \) and \( C \) when \( k = 4 \); and, (2) there are more symbols when \( k = 4 \), therefore, the Cloud needs to assign more storage for the positions when storing swaps in \( S \).

Compression rate for different number of deletions: Fig. \( [8] \) shows the behavior of Bonsai for different number of deletions in the Client side and its effect on the total compression rate. As this figure suggests, increasing the number of deletions in the Client, initially results in Bonsai gaining in terms of total compression rate. This is due to the fact that a larger number of deletions have two effects on the outsourced data. First, it reduces the size of the outsourced data, allowing for more potential deduplications and lower number of required swaps. Second, it alters the probability distribution of the outsourced data more, giving more chance to have a closer probability distribution as the distribution indicated in the policy.

However, after a certain threshold (14 deletions for \( k = 4 \) and 15 deletions for \( k = 8 \)), this trend does not continue and the compression rate stays the same. This is due to the fact that after this threshold, the compression gain on the Cloud is quite low, and is offset by higher storage requirements in
the Client. As the Cloud does not compress $A$ and $S$, which consist of SymbolID and ValueID of the symbols, there is a lower bound on the compression rate achievable by the Cloud. After this point, the compression rate on the Cloud improves this improvement, but it is equal to or less than the loss of compression in the Client, leading to a roughly constant total compression rate. This behavior can be seen for both cases of \( k = 4 \) and \( \text{bits} = 8 \). However, in the case of \( k = 8 \) the compression rate declines faster by increasing the number of deletions, compared to \( k = 4 \); as deletion of the symbols with low probability distribution has a higher effect on reducing the number of required swaps; and shaping the probability distribution of the symbols, leading to more optimal values for SymbolID and ValueID.

**Comparison between Bonsai, Yggdrasil and MKRE:**

Fig. 8 shows the comparison of Bonsai with the closest related work, namely Generalized Deduplication via Multi-Key Revealing Encryption (MKRE) [53], and Dual Deduplication via Yggdrasil [10]. In terms of the global compression rates, as we saw in Fig. 7 Bonsai has a better compression rate for \( k = 8 \) for all datasets used in this work; therefore, this figure only contains the compression rate for \( k = 8 \).

In this Fig. 8 we compare the total compression ratio gained for each dataset using different values of \( n_F \). As illustrated in this figure, the best compression rate achievable by MKRE [53] and Yggdrasil is nearly the same as the compression ratio achieved by Bonsai for HDFS datasets, with Bonsai eventually outperforming both Yggdrasil and MKRE for large files \( k \cdot n_F \geq 2^{12} \). In more homogeneous datasets, such as DVI, MKRE outperforms both Yggdrasil and Bonsai by achieving a compression rate of 0.33 for file size of \( 2^{10} \), compared to 0.82 for Yggdrasil and 0.74 for Bonsai. Whereas Bonsai outperforms MKRE when using datasets with more diversity, such as enron-mail dataset. For example, The difference between compression rate of Bonsai and MKRE for a file size of \( k \cdot n_F = 2^9 \) is 0.26. We note that Bonsai provides information-theoretic security compared to cryptographic security provided by MKRE, giving an edge to Bonsai in many use cases, while maintaining a competitive compression potential.

This figure also suggests that in all datasets, Bonsai outperforms Yggdrasil when the file size is \( k \cdot n_F \geq 2^{10} \). In lower chunk sizes, Yggdrasil outperforms Bonsai in both HDFS logs and enron-mail dataset, yggdrasil outperforms Bonsai for a file size of \( k \cdot n_F \leq 2^{10} \). However, unlike Bonsai, Yggdrasil finds potential deduplications by brute-force search, which introduces huge time complexity for large amounts of data.

Fig. 9 provides a more detailed side-by-side comparison...
of the best achievable compression rate on the HDFS dataset using different methods with file size of $n_F = 256$ and symbol size of $k = 8$. In this case, Yggdrasil achieves the best compression rate on the Cloud (0.6312%), with the expense of high storage requirements on the Client (0.18199). MKRE achieves the best total compression rate of 0.7561 between the three methods that provide privacy, i.e., Yggdrasil (0.8131) and Bonsai (0.7668). We note that Bonsai holds the middle ground by providing competitive compression rate, both in terms of total compression rate and the compression rate on the Cloud. We also note that the privacy provided by Bonsai holds without any cryptographic assumptions. As this figure suggests, Broti outperforms all deduplication schemes by achieving a compression rate of 0.3517. However, this is expected as the DVI dataset is a highly compressible dataset with a low number of duplicate files, which works well with traditional compression techniques as opposed to deduplication schemes.

**Privacy:** Fig. [11] shows the amount of leakage for a strong adversary and weak adversary when the PRNG has been broken. This figure suggests that Bonsai leaks significantly less information to a strong adversary compared to a weak adversary. As a practical example, for $k = 8$ and $n_F = n_{F'} = 15$, Leakage to $A_{\text{strong}}$ is 0.3963 compared to 0.94 leakage to $A_{\text{weak}}$. This is due to the fact that the strong adversary already has a lot of information about the data before receiving the outsourced file $F'$. We note that in a real-life scenario, the adversary is somewhere between the $A_{\text{week}}$ and the $A_{\text{strong}}$.

Fig. [12] shows the uncertainty metric for the same variables. This figure includes the uncertainty for $k = 8$, where $A_{\text{strong}}$ and $A_{\text{week}}$ are considered in two scenarios, where (a) the adversary has broken the PRNG, and (b) the adversary cannot break the PRNG in polynomial time. We choose $k = 8$ as a Byte-sized symbol is more practical in real-life scenarios. Fig. [12] shows that even the strong adversary with a broken PRNG faces a large uncertainty to guess the real file $F$. As an example, deleting 15 symbols or 15 Bytes of data leads to an average of 95.02 Shannon bits of entropy for $A_{\text{strong}}$ with negligible probability of breaking the PRNG, and 83.40 bits of Shannon entropy for the said $A_{\text{strong}}$ if it breaks the PRNG. These shannon bits of entropy when the Client performs as low as 9 deletions, are 61.11 when the adversary can not break the PRNG, and 54.11 when the PRNG is broken. As a standard in information-theoretic literature, 49 bits of entropy is considered secure against an adversary with the goal of identifying the original file [54]. We note that the assumption of breaking the PRNG is a very hard assumption for the adversary to fulfill (as it entails guessing the seed used by Client to create $F'$). Our results suggest that Bonsai is still information-theoretically secure even when the output of the PRNG is known to the adversary. This is not surprising, as such information only leaks the deleted positions, but not the deleted values.

Another interesting metric to analyse the privacy of Bonsai is the probability of the original $F$ compared to other possible pre-images of $F'$. In order to assign a scientific understanding to this measure, we calculated the percentage of outsources, where the original file $F$ is among the highest probable $q$ images of the outsourse $F'$. In other words, considering the case where the adversary generates all the possible pre-images of $F'$ and sorts then based on their probability, assuming that the adversary can pick the first $q$ chunks in this sorted list, what is the probability that the original file $F$ is among the selected
Similarly to Yggdrasil, Bonsai’s privacy relies on deletions in the cloud with higher probability. The function to ensure that the generated base can be deduplicated by a PRNG. This behavior mimics a deletion channel. We implemented a filtering method using multiple seeds of a PRNG to improve compression, e.g., using more compact representations for the sorting step, improve privacy, e.g., combine random deletions with value changes or insertions at the Client side, and efficient secure sharing mechanisms with third parties.

VII. CONCLUSION

This paper proposes Bonsai, a privacy-aware dual deduplication system that preserves the privacy of Client’s data while achieving attractive compression for cloud storage providers. Similarly to Yggdrasil, Bonsai’s privacy relies on deletions of symbols from the original data; however, Bonsai achieves significantly better compression rates on the client side by using the proposed approach provides good privacy guarantees: our analysis shows that an attacker faces a high degree of uncertainty when trying to reconstruct Clients’ original data from what is outsourced. In real-life scenarios, The strongest possible adversary faces an average 83.40 Shannon bits of entropy while trying to guess the original data, even if the adversary has full knowledge about the characteristics and statistics of the original data source. We further show that there is only 50% chance that the adversary can guess the correct original data from the received outsourced data, by checking $2^{15}$ possible strings for each chunk of data. Even if the adversary has the possibility to check $2^{32}$ possible strings, the chance of guessing the correct file for each chunk is still less than 90%.

Future work will study additional mechanisms for Bonsai to improve compression, e.g., using more compact representations for the sorting step, improve privacy, e.g., combine random deletions with value changes or insertions at the Client side, and efficient secure sharing mechanisms with third parties.

REFERENCES

[1] P. Deutsch, “Rfc1951: Deflate compressed data format specification version 1.3,” 1996.
[2] J. Gilchrist, “Parallel data compression with bzip2,” in Proceedings of the 16th IASTED international conference on parallel and distributed computing and systems, vol. 16. Citeseer, 2004, pp. 559–564.
[3] J. Alakuijala, A. Farruggia, P. Ferragina, E. Kliuchnikov, R. Obryk, Z. Szabadka, and L. Vandevenne, “Brotli: A general-purpose data compressor,” ACM Transactions on Information Systems (TOIS), vol. 37, no. 1, pp. 1–30, 2018.
[4] D. Korn and K. Vo, “The vdfiff generic differing and compression data format,” Work in Progress, 2002.
[5] K. Jin and E. L. Miller, “The effectiveness of deduplication on virtual machine disk images,” in Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference, ser. SYSTOR ’09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1534530.1534540
[6] W. Xiu, H. Jiang, D. Feng, F. Douglas, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, “A comprehensive study of the past, present, and future of data deduplication,” Proceedings of the IEEE, vol. 104, no. 9, pp. 1681–1710, 2016.
[7] C. Constantinescu, J. Glider, and D. Chambless, “Mixing deduplication and compression on active data sets,” in 2011 Data Compression Conference. IEEE, 2011, pp. 393–402.
[8] R. Vestergaard, D. E. Lucani, and Q. Zhang, “Generalized Deduplication: Lossless Compression for Large Amounts of Small IoT Data,” in European Wireless Conf., may 2019.
[9] R. Vestergaard, Q. Zhang, and D. E. Lucani, “Generalized Deduplication: Bounds, Convergence, and Asymptotic Properties,” in IEEE GLOBECOM, dec 2019.
[10] H. Sehat, E. Pagnin, and D. E. Lucani, “Yggdrasil: Privacy-aware dual deduplication in multi client settings,” pp. 1–6, 2021.
[11] M. Bellare, S. Keelvedhi, and T. Ristenpart, “Message-locked encryption and secure deduplication,” in Annual international conference on the theory and applications of cryptographic techniques. Springer, 2013, pp. 296–312.
[12] K. Akhila, A. Ganesh, and C. Sunitha, “A study on deduplication techniques over encrypted data,” Procedia Computer Science, vol. 87, pp. 38–43, 2016.
[13] D. E. Lucani, L. Nielsen, C. Orlandi, E. Pagnin, and R. Vestergaard, “Secure generalized deduplication via multi-key revealing encryption,” in International Conference on Security and Cryptography for Networks. Springer, 2020, pp. 298–318.

In addition, Bonsai protects the confidentiality of Clients’ data thanks to a deletion-channel-like approach. We argue the proposed approach provides good privacy guarantees: our analysis shows that an attacker faces a high degree of uncertainty when trying to reconstruct Clients’ original data from what is outsourced. In real-life scenarios, The strongest possible adversary faces an average 83.40 Shannon bits of entropy while trying to guess the original data, even if the adversary has full knowledge about the characteristics and statistics of the original data source. We further show that there is only 50% chance that the adversary can guess the correct original data from the received outsourced data, by checking $2^{15}$ possible strings for each chunk of data. Even if the adversary has the possibility to check $2^{32}$ possible strings, the chance of guessing the correct file for each chunk is still less than 90%.

![Figure 13](image-url)
