On Information Leakage in Deduplicated Storage Systems

Hubert Ritzdorf
Department of Computer Science
ETH Zurich, Switzerland
hubert.ritzdorf@inf.ethz.ch

Ghassan O. Karame
NEC Laboratories Europe
Heidelberg, Germany
ghassan@karame.org

Claudio Soriente
Telefónica Research
Barcelona, Spain
claudio.soriente@telefonica.com

Srdjan Čapkun
Department of Computer Science
ETH Zurich, Switzerland
srdjan.capkun@inf.ethz.ch

ABSTRACT

Most existing cloud storage providers rely on data deduplication in order to significantly save storage costs by storing duplicate data only once. While the literature has thoroughly analyzed client-side information leakage associated with the use of data deduplication techniques in the cloud, no previous work has analyzed the information leakage associated with access trace information (e.g., object size and timing) that are available whenever a client uploads a file to a curious cloud provider.

In this paper, we address this problem and analyze information leakage associated with data deduplication on a curious storage server. We show that even if the data is encrypted using a key not known by the storage server, the latter can still acquire considerable information about the stored files and even determine which files are stored. We validate our results both analytically and experimentally using a number of real storage datasets.

1. INTRODUCTION

With the ever increasing amount of data produced worldwide, the cloud offers a cheaper and more reliable alternative to local storage. Existing cloud service providers such as Amazon S3, Microsoft Azure, or Dropbox guarantee a good trade-off between quality of service and cost effectiveness. The cloud has also gained many clients among SMEs and large businesses that are mainly interested in storing large amounts of data while minimizing the costs of both storage and infrastructure management/maintenance.

Existing cloud solutions store duplicate data uploaded by different users only once—thus saving storage costs. Recent studies show that cross-user data deduplication can reduce storage costs by more than 40% in standard file systems and by up to 83% in back-up applications [15].

The literature features a number of proposals for securing data deduplication (e.g., [1, 3, 6, 17, 18, 20, 21]) in the cloud. All these proposals share the goal of enabling cloud providers to deduplicate *encrypted* data stored by their users. Such solutions allow clients to reduce storage costs, while they ensure confidentiality of the stored data. These works, however, do not consider the information leakage incurred by the access traces. We assume that each entry in an access trace contains a timestamp, an object ID, and the object size.

Previous work has analyzed client-side information leakage associated with data deduplication. For example, Harnik *et al.* [10] describe how a malicious client can learn whether a file is already stored on a particular cloud by guessing predictable content hashes. This leakage can be countered using Proof of Ownership (PoW) schemes [5, 9], which enable clients to prove possession of a file. However, PoW can only prevent information leakage towards malicious clients, and not towards the storage provider itself (since the provider is the one checking the PoW). To the best of our knowledge, no prior work has analyzed the information leakage associated with the access traces that are available to the service provider.

In this paper, we analyze the information leaked to a curious storage provider by systems that perform client-side deduplication and encryption. We assume that the underlying client-side encryption is secure, and we show that the storage provider can still acquire considerable information about the stored files *without knowledge of the encryption key*. Our results show that data deduplication offers a strong distinguisher for a curious storage server to determine which files are stored. We confirm our analysis by means of thorough experiments using real datasets approximately amounting to 13.5 TiB of data. Finally, we discuss the solution space to minimize information leakage associated with existing data deduplication techniques. Our work therefore lays the foundations for quantifying the trade-off between security and storage-efficiency associated with various data deduplication techniques.

Our contributions in this paper are summarized as follows:

- We analyze and quantify the information leakage to a curious storage server due to data deduplication. Our results show that popular content-based chunking methods offer clear distinguishers for stored content, even when such content is securely encrypted. In particular, a curious storage provider can probabilistically determine the absence and presence of stored files by observing the size of deduplicated content. Although such leakage is also witnessed in fixed-sized block-based and file-based deduplication, we show that the prediction of a curious storage provider is more accurate for content-defined chunking (CDC)-based deduplication.
- We validate our analysis by means of experiments using real datasets. We extract two novel datasets from storage servers of an industrial research lab and a university. Our results show that existing CDC-based deduplication schemes leak information by up to 54%.
- We explore the solution space to limit information leakage associated with existing block-based data deduplication techniques. Our findings suggest that little can be done to prevent leakage due to CDC-based schemes. In this respect, we show that fixed-sized block-based deduplication technologies establish a solid trade-off between the achieved privacy and the storage-efficiency performance.

The remainder of the paper is organized as follows. In Section 2, we briefly overview existing data deduplication techniques. In Section 3, we describe our model and present our privacy metrics. In Section 4, we analyze information leakage due to existing data deduplication techniques. In Section 5, we validate our analysis by means of evaluation using real datasets. In Section 6, we discuss further insights with respect to our analysis and explore the solution space to minimize information leakage due to data deduplication. In Section 7, we overview related work in the area, and we conclude the paper in Section 8.

2. BACKGROUND

Deduplication techniques mostly fall into two broad categories: file-based and block-based techniques [16]. All existing techniques, however, rely on indexing in order to identify already deduplicated content and to reconstruct the original files. As we discuss in this section, the indexing overhead greatly varies depending on the underlying deduplication technique.

In order to explain the differences amongst the deduplication techniques considered in this paper, we rely in the sequel on a running example with 5 files $F_1, .., F_5$, as depicted in Figure 1. Files F_1 and F_2 are identical, F_3 extends F_1 with a prepended byte (each byte of the files in Figure 1 is depicted by a character), F_4 extends F_1 with a few appended bytes, and finally F_5 is identical to F_1 in all but the last byte. For the deduplicated storages in Figures 1b - 1d, we depict the stored data objects on the left and the indices used to reconstruct files from the objects on the right.

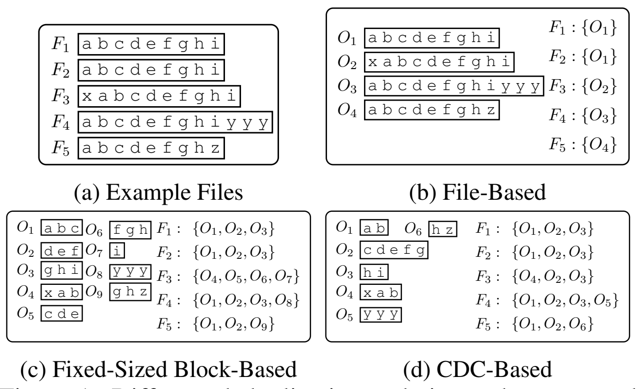


Figure 1: Different deduplication techniques demonstrated for five example files $F_1 .., F_5$. The original storage layout is depicted in 1a. Figures 1b - 1d show the deduplicated storage including the stored data objects and the indices for the different techniques.

2.1 File-based Deduplication

File-based deduplication only deduplicates identical files. Deciding whether two files are identical is usually achieved by hashing the contents of each file and comparing the results. In the example of Figure 1b, F_1 is actually stored, while F_2 is deduplicated and simply a pointer to F_1 is maintained (instead of storing the whole F_2). File-based deduplication requires modest computational and indexing overhead. The main drawback of file-based deduplication is that it does not result in any storage savings if two files differ even in a single bit (since the resulting hash would be different). For example, in Figure 1b, F_3, F_4 and F_5 are all stored, even though they share a large amount of content with F_1.

2.2 Block-based Deduplication

Block-based deduplication chunks a file into blocks and deduplicates any two blocks with identical content. This technique enables fine-grained deduplication and overcomes the drawbacks of file-based deduplication.

The simplest chunking algorithm splits a file in blocks of fixed size. Fixed block-size chunking can efficiently deduplicate files that only differ in one or a few blocks. In Figure 1c, only the blocks of F_4 and F_5 that differ from the ones of F_1 are stored; blocks in common with F_1 are referenced to with a pointer. Small block sizes may increase the storage savings (since the probability that two blocks are identical increases)—this however comes at the expenses of larger indexes. Previous work has shown that block sizes of 4 to 8 KiB yield the best storage savings taking into account the savings for deduplication and the size of the index [14, 15].

Fixed block-size chunking, however, fails to effectively deduplicate even slightly "shifted" content. For instance, in

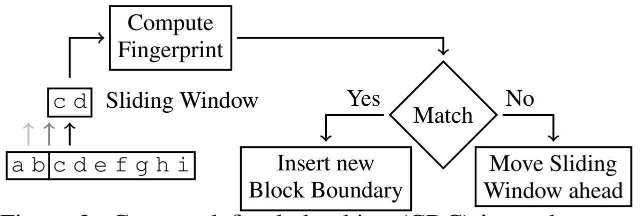


Figure 2: Content-defined chunking (CDC) is used to produce variable-sized blocks, well-suited for deduplication. In CDC, a sliding window is moved over a file and block boundaries are inserted depending on the computed fingerprints.

Figure 1c all blocks of F_3 are stored because they are all different from the ones of F_1, even thought F_3 has the same content of F_1 with a prepended byte. This shortcoming can be effectively addressed by content-defined chunking (CDC) algorithms.

CDC produces variable-sized blocks by processing files with a sliding window with one-byte steps. At each offset, CDC computes a fingerprint of the content in the window and inserts a block boundary at the end of the window if the fingerprint matches a pre-defined value; if a block boundary is inserted at byte i, the new windows starts at byte $i{+}1$. This procedure is depicted in Figure 2 where only the first block of F_3 that contains the prepended byte, has to be stored. The remaining blocks can just be referenced with a pointer to the stored blocks of F_1.

Popular functions for CDC algorithms consist of rolling checksums, e.g., based on Rabin fingerprints [14, 16, 19]. Rolling checksums allow the efficient computation of checksums based on a sliding window, as the checksum of the current window can be efficiently computed using the checksum of the previous window. For example, Rabin Fingerprints use a window of n bits $(a_1, ..., a_n), a_i \in \mathbb{Z}_2$ as coefficients of the polynomial $p(x) = a_1 * x^{n-1} + ... + a_n$. The fingerprint is then defined as $r(x) = p(x) \bmod q(x)$, where $q(x)$ is an irreducible polynomial. CDC places a boundary at the end of the current window, if the last l bits of $r(x)$ match the last l bits of a pre-defined constant. Tuning the value of l allows for controlling the expected block size (a small l leads to more matches and smaller blocks, while a large l features less matches and hence larger blocks). The fingerprint for the current window can be efficiently computed by "subtracting" the first byte of the previous window and "adding" the last byte of the current window. CDC usually defines a minimum and a maximum block size in order to avoid very small or very large blocks [7].

As shown in Figure 1c, CDC successfully deduplicates most of the blocks of F_3.

2.3 Deduplication over Encrypted Data

Since data is often outsourced to the cloud, (client-side) encryption is clearly desirable to protect data confidentiality with respect to third parties and a curious storage server.

Clearly, encryption comes at odds with deduplication; encrypting two identical files (or blocks) with a semantically secure cipher, leads to two different ciphertexts with overwhelming probability. Deterministic encryption only helps in a single-client scenario where the client uses the same encryption key to encrypt any of his files before uploading them to the storage server. However, the largest savings of deduplication typically originate from cross-user deduplication, i.e., when files from different users are deduplicated.

One approach to enable cross-user deduplication over encrypted data is to ensure that whenever a given file (or block) is encrypted by any user, the same encryption key is used, so that the output ciphertext is always the same. This approach is taken by Message-locked Encryption (MLE) [4] and its most prominent instantiation being Convergent Encryption (CE) [6]. A CE encryption scheme uses deterministic symmetric encryption and sets the encryption key for a file to be the hash of the file content. Therefore, the encryption of a given file always results in the same ciphertext. However, CE is vulnerable to off-line dictionary attacks. In particular, the adversary can "guess" the file content, CE-encrypt it, and match the computed ciphertext against the stored one [10]. If CE is used in a block-based deduplication system, a guessing attack can be mounted against single file blocks.

To mitigate a guessing attack against CE, systems like DupLESS and ClearBox [1, 3] use a semi-trusted key server as a third party. Here, clients engage in an oblivious protocol with the key server to generate encryption keys for their files. The protocol ensures that the key server does not learn the content submitted by a client, while enabling any two clients that have identical files to receive the same encryption key. The key server makes guessing attacks impractical as it introduces rate limitation. As long as the key server is not compromised, DupLESS and ClearBox provide semantic security for the encrypted data; in case the key server is compromised, the security provided by these schemes falls back to that of CE.

Notice that Dupless and ClearBox assume file-based deduplication but could easily be adapted to work with block-based deduplication. Clearly, if block-based deduplication techniques are employed, chunking must be applied before encryption. If encryption would precede chunking, slightly modified files would result in largely different ciphertexts. Given two such ciphertexts as input, the chunking algorithms will not likely find shared content to deduplicate.

3. MODEL

In this section, we introduce our system and threat model, our storage graph model, and our privacy metrics.

3.1 System Model

In the sequel, we use the term *object* to refer to the unit of deduplication. For instance, in file-based deduplication, objects correspond to whole files; in case of block-based deduplication, objects refer to blocks output by the chunking algorithm.

Similar to [1, 3], we assume a storage server $\mathcal{S}$ and a number of clients. We assume that the chunking/encryption operations are performed at the client. Indeed, client-side deduplication is widely used amongst cloud storage services,

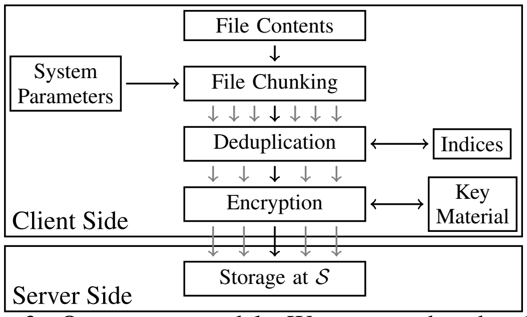


Figure 3: Our system model. We assume that the clients perform chunking, intra-file deduplication, and encryption before uploading the contents to $\mathcal{S}$. We assume the use of strong encryption keys that are kept secret from $\mathcal{S}$. Therefore, the adversary $\mathcal{A}$ can only observe ciphertexts and their sizes.

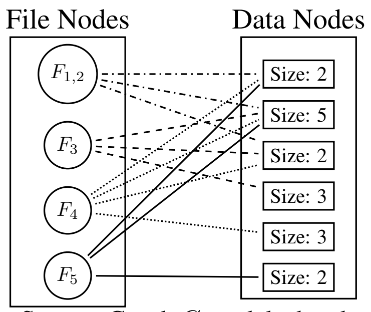


Figure 4: The Storage Graph G models the observable structure of $\mathcal{S}$. To create G, we deduplicate the files $F_1.., F_5$ from Figure 1 using CDC-based deduplication. Note that $\mathcal{S}$ cannot distinguish F_1 and F_2. Therefore $\mathcal{S}$ only observes four files. The data vertices in G are connected to the file vertices they belong to.

such as Dropbox, since it results in considerable bandwidth savings. We also note that client-side deduplication reduces information leakage towards the server when compared to the case where the server orchestrates deduplication; for example, the server does not learn about redundant blocks contained in a single file. Recall that in client-side deduplication systems, the storage server does not perform data chunking and is not equipped with the necessary keys to encrypt/decrypt file objects. This abides by the end-to-end security argument in which the encryption keys are retained by the clients.

As shown in Figure 3, we assume that a client initially chunks the file contents according to a system-wide chunking algorithm; if file-based deduplication is configured, no chunking is required. Next, the client encrypts the resulting objects using a secure block cipher and a secret key, and uploads the corresponding ciphertexts to the cloud storage server $\mathcal{S}$. Here, we assume that the utilized encryption function is length-preserving.

Similar to DupLESS or ClearBox [1, 3], the encryption key can be generated using the help of a key server (which is independent of the storage server). We also assume that file metadata, such as filenames, are not visible to $\mathcal{S}$ (e.g., the filenames are also encrypted).

3.2 Threat Model

We consider an adversary (denoted by $\mathcal{A}$) that is interested in learning whether a given *plaintext* file is stored at $\mathcal{S}$. For instance, the adversary could be a state agency that is interested in checking whether a given copyrighted document is stored in the cloud. $\mathcal{A}$ can acquire the access traces of $\mathcal{S}$ (e.g., through compromise, coercion, subpoena power). We assume that entries in such an access trace contain a timestamp, the object ID, and the object size. By leveraging the access traces, $\mathcal{A}$ can try to identify which objects form a file[1] (e.g., by correlating timestamps).

As mentioned earlier, we further assume that all stored contents are encrypted, the encryption keys are unknown to $\mathcal{A}$ and cannot be guessed by the latter. We further assume that $\mathcal{A}$ is computationally bounded and, as such, cannot break secure encryption functions. Finally, we assume that the clients do not collude with the adversary. Notice that a malicious client could trivially check whether certain files have been stored on $\mathcal{S}$ by uploading these files and observing the changes on $\mathcal{S}$ [10].

To summarize, we consider the "ideal" setting for secure deduplication storage systems, where the storage server cannot guess or acquire the encryption keys, and can only observe limited information (such as objects size and timestamp or storage indexes) which is revelead whenever clients upload contents to the cloud. We show that, even in such settings, data deduplication leaks considerable information about the stored contents.

3.3 Storage Graph

We model any file f as a tree $T(f)$. Each leaf node represents an object of f as output by the chunking algorithm; in case of file-based deduplication, the chunking algorithm outputs only one object. Leaf nodes also have an attribute *size* that is set to the size of the corresponding object. All leaves of a file are connected to a common root node.

The set of files on $\mathcal{S}$ is modeled with a graph $G = (V, E)$ populated with the trees of the stored files; as it will become clear, trees added to G may share some leaves.

G is initially empty and trees are added as follows whenever a new file f is stored. If f can be entirely deduplicated, we leave G unchanged. Otherwise, we create a new tree $T(f)$, initially comprising only the root node. The leaves of $T(f)$ depend on which objects of f can be deduplicated and which must be actually stored on $\mathcal{S}$. If an object of f cannot be deduplicated, a new leaf node is created and linked to the root of $T(f)$. If an object of f can be deduplicated, i.e., it is identical to an object already stored on $\mathcal{S}$, the root of $T(f)$ is connected to the node of G that represents the stored object.

Given the above model, each root node of G is connected to exactly one leaf node in case of file-based deduplication. In case of block-based deduplication, each root is connected to at least one leaf node. If files are chunked in blocks using a fixed block-size, most leaves have the same size (since only the last block of a file may have a size smaller than the block-

[1]This information can be alternatively acquired by compromising the storage index that is maintained by $\mathcal{S}$.

size parameter); if chunking is based on file content, leaves may have different sizes, as seen in Figure 4.

Notice that G is built by examining at the access traces of the storage server, without the need to learn the contents of the stored files. Therefore, G can be built even if the contents are encrypted with unknown keys. We also point out that G is a bipartite graph where the set of nodes is partitioned into file nodes and object nodes.

3.4 Anonymity and Candidate Sets

For any file f, we denote its *deduplication fingerprint* by $T(f)$. Given two files f, f', we say that they have the same deduplication fingerprint if $T(f)$ is a valid isomorphism of $T(f')$. An isomorphism between the two trees is valid if it preserves nodes, edges, and the sizes of the leaves. For example, the deduplication fingerprint of file F_1 in Figure 4 is the subtree of G with nodes $F_{1,2}$, Size: 2, Size: 5 and Size: 2, since the file comprises three chunks of sizes 2, 5, and 2, respectively. We stress that a deduplication fingerprint, cannot be used to uniquely identify files or objects (unlike cryptographic hashes) since it only contains sizes (and not file IDs); in our example, files F_1, F_2 and F_5 share the same deduplication fingerprint but not the content.

The *anonymity set* of f is the set of *all possible files* that have the same deduplication fingerprint as f. Clearly, "the set of all possible files" is application-dependent and, in the worst case, amounts to all binary strings of a given length.

Similarly, the *candidate set* of f in G (denoted as $C(f, G)$) is the set of files stored on $\mathcal{S}$— and hence represented in G— that have the same deduplication fingerprint as f.

In order to compute $C(f, G)$, we proceed as follows. We start with $C(f, G) = \emptyset$. We go through each root node of G, identify its tree $T(f')$, and check if the deduplication fingerprint of f' is equal to the one of f. In this case, we add f' to $C(f, G)$ and remove $T(f')$ from G before moving to the next root node. Notice that leaf nodes of $T(f')$ that belong to other trees are not removed. If $T(f')$ is not a valid isomorphism of $T(f)$, we keep moving to the next root node until all root nodes have been examined.

If $f' \in C(f, G)$, then we say that f' is a *candidate* for f. In case $f' \neq f$, we say that we have a deduplication fingerprint *collision*.

Notice, that $\mathcal{A}$ can construct the storage graph G, find deduplication fingerprints for stored files and thereby compute the candidate set $C(f, G)$ for any file f on $\mathcal{S}$. None of these actions require knowledge of the encryption key.

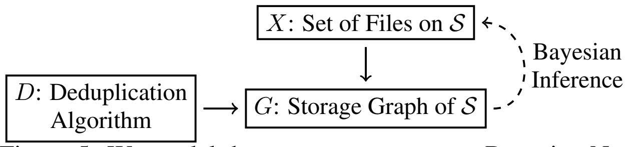
Where appropriate, we additionally consider the probability that f is stored on $\mathcal{S}$—which we denote by Probability of Storage (PoS). Namely, PoS quantifies the probability that at least one of the stored candidates indeed corresponds to f.

4. STORAGE INFERENCE

In this section, we proceed with analyzing the probability that a target file f is stored on $\mathcal{S}$ given the storage graph G.

4.1 File Absence

We start with the following observation. *If candidate set $C(f, G) = \emptyset$, the adversary is certain that f is not stored on* $\mathcal{S}$. This is due to the fact that if f would have been stored, the storage graph G would contain at least one valid isomorphism for $T(f)$.



X: Set of Files on $\mathcal{S}$
D: Deduplication Algorithm
G: Storage Graph of $\mathcal{S}$
Bayesian Inference

Figure 5: We model the storage process as a Bayesian Network. $P(X, G, D) = P(G|X, D) \cdot P(X) \cdot P(D)$

Clearly, f might have been compressed or encoded before being uploaded to the cloud. However, compression and encoding are public functions and therefore we assume that the adversary is aware of this transformation.

Checking for file absence can have serious implications on existing outsourced storage systems that rely on deduplication. For instance, if an enterprise is required by law to store certain files, an external auditor can directly prove that the files have not been entirely stored simply by observing the storage graph, and without having access to the necessary keys to decrypt data.

4.2 File Presence

If $C(f, G) \neq \emptyset$, the candidate set defined in the previous section is not sufficient to quantify the probability that f is actually stored on $\mathcal{S}$.

In order to reason about the presence of f on $\mathcal{S}$, we must take into account how likely it is for any file $f' \neq f$ to have $T(f') = T(f)$, i.e., we must also consider the anonymity set. If the size of the candidate set is identical to that of the anonymity set, it means that *all* possible files that share its deduplication fingerprint with f (including f itself) are stored on $\mathcal{S}$. Otherwise, if the size of the candidate set is smaller than the one of the anonymity set we cannot conclude that f is stored on $\mathcal{S}$ because we may just be witnessing deduplication fingerprint collisions.

Determining the anonymity set of the target file requires knowledge of the *storage domain*, i.e., the set of all possibly stored files. Knowledge of the complete storage domain is a strong assumption in general-purpose storage systems but may be justified if the storage server only hosts specific types of files. Contextual information may also increase the chances of $\mathcal{A}$ knowing the storage domain. For example, if the target file is a movie and files on $\mathcal{S}$ are organized in folders, then the adversary may only take into account the files stored in the "video" folder.

Given a file f, the file domain, and the storage graph G, we now compute the probability that f is on stored $\mathcal{S}$— which we denote by Probability of Storage (PoS). To do so, we model the state of $\mathcal{S}$ as a Bayesian Network [12], as depicted in Figure 5. The Bayesian Network consists of three random variables. Variable X represents the set of all files stored on $\mathcal{S}$. The probability distribution of X depends on file popularity and possible storage combinations. For instance, a popular movie is a likely storage candidate and different photos of the same person are also likely stored together. D is the random variable capturing the deduplication algorithm used on $\mathcal{S}$. Usually, D is known, but our model also accounts for potential uncertainty. Finally, G is the random variable describing the storage graph of $\mathcal{S}$. G is conditioned by D and X, since the storage graph depends on both of these variables.

To learn whether $f \in X$, we assume knowledge of the deduplication algorithm D_o and the observed storage graph G_o. Using this knowledge, we can determine the Probability of Storage as:

$$\text{PoS}(f, G_o, D_o) = P(f \in X|\, G = G_o, D = D_o)$$

We compute the Probability of Storage for a given file f using Bayesian inference as follows. We apply bottom-up reasoning where we observe evidence for D and G and use the evidence to infer the updated probabilities $P(X)$. Based on the probabilities $P(X)$, we compute $P(f \in X)$. Given the observed deduplication algorithm D_o and the observed storage graph G_o, we compute $P(f \in X)$ as:

$$\begin{aligned}\text{PoS}(f, G_o, D_o) &= P(f \in X|\, G = G_o, D = D_o)\\ &= \frac{\sum\limits_{X_p : f \in X_p} P(X = X_p, G = G_o, D = D_o)}{\sum\limits_{X_p} P(X = X_p, G = G_o, D = D_o)}\end{aligned} \quad (1)$$

That is, to determine PoS, we sum over all possible file sets X_p. Following the properties of the Bayesian Network, the joint probability $P(X, G, D)$ is computed as:

$$P(X, G, D) = P(G|X, D) \cdot P(X) \cdot P(D)$$

Notice that the knowledge of $P(X)$ is only required for those values of X where $P(G|X, D) > 0$. That is, we only need to know the probability that a certain set of files X is stored on $\mathcal{S}$, if the deduplication fingerprints of X match the storage graph G given the deduplication algorithm D.

4.2.1 Template Attacks

So far, we showed how to compute the Probability of Storage for single files given knowledge of the storage domain and the file popularity. We now provide a concrete example where such knowledge is available to an adversary. We frame this scenario as a *template attack*. In this attack, we leverage the Probability of Storage of different files in order to infer higher-level contextual information.

Template attacks assume that $\mathcal{S}$ stores a number of files that originate from a common template. For instance, consider the scenario where an enterprise stores employment contracts. All contracts are generated from a common template where the general conditions are fixed, while the name and the salary are filled in for each employee.

We assume the adversary $\mathcal{A}$ knows the fixed parts of the template and tries to infer the variable parts of the stored contracts. We model the content of the contracts with two categories of random variables: *Target Variables* are the random variables that represent valuable information for the adversary (i.e., names and salaries of the employees). *Marginal Variables* are random variables that model the information that is not of interest for the adversary. For example, if contracts are stored as PDFs, they contain creation timestamps [11].

$\mathcal{A}$ tries to link the target variables, such as name and salary. To do so, $\mathcal{A}$ tests the probability of a number of combinations $\mathcal{C}$, e.g., John Doe earns \$50,000. To compute this, $\mathcal{A}$ identifies all files from the storage domain with this combination which we denote by $\mathcal{F}_\mathcal{C}$, e.g., $\mathcal{F}_\mathcal{C}$ are all files with "John Doe" and "\$50,000". Then, $\mathcal{A}$ determines the Probability of Storage for all files in $\mathcal{F}_\mathcal{C}$ and all trees T_s that can be found in G.

Finally, $\mathcal{A}$ computes the probability that at least one file from $\mathcal{F}_\mathcal{C}$ is stored on $\mathcal{S}$:

$$P(\mathcal{C}, G, D) = 1 - \prod_{T_s \in G} (1 - \sum_{f_i \in \mathcal{F}_\mathcal{C}} \text{PoS}(f_i, T_s, D))$$

In Section 5.4, we evaluate template attacks and their accuracy against different deduplication schemes.

We acknowledge, however, that the assumption that the adversary knows all possible sets of files that could be stored on $\mathcal{S}$ and their respective probability cannot always be met. In the following section, we analytically show that existing data deduplication schemes still leak considerable information towards the storage server, even in scenarios where such assumptions cannot be met.

4.3 Quantifying Anonymity Sets

Recall that the anonymity set of a file f is the set of all possible files that have the same deduplication fingerprint as f. That is, $A(f) = \{f' \in \{0,1\}^* :\ T(f') \simeq T(f)\}$. This set clearly depends on the deduplication algorithm used by the storage server. The cardinality of an anonymity set for file f refers to the number of files in the storage domain that share the deduplication fingerprint with f.

In the following, we analyze the cardinality of the anonymity set for a given file f of size n bytes, with respect to each of the deduplication techniques considered in this paper.

4.3.1 File-Based Fingerprints

In file-based deduplication, a file of size n bytes can have exactly one deduplication fingerprint, i.e., a tree with one leaf node of size n. In other words, all files of size n share the same deduplication fingerprint (regardless of their content). Therefore, the anonymity set $A(f) = \{0,1\}^n$ and its cardinality is 2^n.

4.3.2 Fixed-Sized Block-Based Fingerprints

Fixed-sized block-based deduplication splits a file of n bytes into $\lceil \frac{n}{B} \rceil$ blocks, where B is the chosen block size. All blocks but the last have size B; the last block has size $n \mod B$. Notice that since we assume client-side deduplication, identical blocks are not included in the fingerprint. That is, if a file has two identical blocks they are deduplicated at the client and only one is uploaded to $\mathcal{S}$.

Consequently, if f has x B-sized blocks, the anonymity set of f comprises all files whose trees have x leaves of size B and $\lceil \frac{n}{B} \rceil - \lfloor \frac{n}{B} \rfloor$ leaves of size $n \mod B$.

4.3.3 CDC-Based Fingerprints

In contrast to the previous schemes, a file of size n can have a large number of possible CDC-based deduplication fingerprints. This is due to the fact that the block sizes could be anywhere between a minimum block size α and a maximum block size β. This results in the fact that the anonymity set for each file shrinks—resulting in additional leakage per deduplication fingerprint.

To quantify the associated information leakage, we provide a lower bound for the number of possible deduplication fingerprints for a given file size n, such that the sum of the sizes of their leaves totals n. We first compute the number of possible fingerprints given file size n, k blocks and a maximum block size β using an modified version of the stars-and-bars theorem [8, 22]:

$$S(n,k,\beta) = \sum_{q=0}^{\min(k,\frac{n}{\beta+1})} (-1)^q \cdot \binom{k}{q} \cdot \binom{n-q(\beta+1)+k-1}{k-1}$$

We extend this to also include the minimum block size α:

$$\begin{aligned} S(n,k,\alpha,\beta) =& \frac{S(n-k\cdot\alpha,k,\beta-\alpha)}{k!} \\ &+ \sum_{r=1}^{\alpha-1} \frac{S(n-(k-1)\cdot\alpha-r,k-1,\beta-\alpha)}{(k-1)!} \end{aligned}$$

Finally, we iterate over the possible number of blocks. The minimum number of blocks is given by $\lceil n/\beta \rceil$, while the maximum number of blocks is given by $\lceil n/\alpha \rceil$. Therefore, the lower bound for the number of CDC-based deduplication fingerprints is:

$$N(n,\alpha,\beta) = \sum_{k=\lceil n/\beta \rceil}^{\lceil n/\alpha \rceil} S(n,k,\alpha,\beta) \qquad (2)$$

As an example, consider a CDC scheme with an average block size of 4 KiB (α = 2KiB, β = 8KiB). For files of size 8 KiB, there are 242,984,335 possible deduplication fingerprints. In this case, the lower bound as computed from Equation 2 is N(8KiB, 2KiB, 4KiB) = 209,014,500 possible fingerprints—thus demonstrating the rapid growth of the number of CDC-based deduplication fingerprints and the validity of our lower bound.

In the following, we show that two randomly chosen files of the same size are unlikely to share a deduplication fingerprint when CDC-based deduplication is used. Namely, we analyse the probability that two randomly chosen files (of the same size) have identical deduplication fingerprints.

To analytically determine the collision probability, we need the number of possibilities $S(n,k,\alpha,\beta)$ to split a file of size n into k blocks of minimum size α and maximum size β.

We model the CDC algorithm using two variables: P_h is the probability of inserting a boundary and P_β is the probability of creating a maximum-sized block. We set $P_\beta = (1-P_h)^{\beta-\alpha}$, which represents the case where a boundary was not inserted $(\beta-\alpha)$ times. We now compute the probability $P(n,k,m)$ of fingerprint collisions for k blocks, m of which are maximum-sized:

$$P(n,k,m) = ({P_\beta}^m \cdot {P_h}^{k-m} \cdot (k-m)!)^2 \cdot S(n,k,\alpha,\beta)$$

	Dataset 1	Dataset 2	Dataset 3	Dataset 4
Original Size	113.6 GiB	13.4 TiB	1.54 TiB	30.9 GiB
# Files	288,906	52,634,346	3,581,951	900,000
# Unique Files	219,578	13,646,320	1,696,978	900,000
Block Sizes	4, 8 KiB 1 MiB	8, 16 KiB 32, 64 KiB	4, 8 KiB 1 MiB	4, 8 KiB 1 MiB

Table 1: The datasets used for experimental evaluation.

Given this, we now compute the probability of fingerprint collisions for two random files of size n consisting of k blocks. In the case where $k > 2$, we iterate over the number of maximum-sized blocks. Otherwise, if $k = 2$ we approximate the collision based on the position of the only boundary. If $k = 1$, there will be a collision since both files will have the signature n.

$$P(n,k) = \begin{cases} 1 & \text{if } k = 1 \\ ((1-P_h)^{(k-\alpha)})^2 & \text{if } k = 2 \\ \sum_{m=0}^{\lfloor \frac{n}{\beta} \rfloor} P(n,k,m) & \text{if } k > 2 \end{cases}$$

Finally, we can compute the overall collision probability for two randomly selected files of size n by computing

$$\text{CDC-Collision}(n) = \sum_{k=\lceil \frac{n}{\beta} \rceil}^{\lceil \frac{n}{\alpha} \rceil} P(n,k)$$

In Section 5.3, we confirm this analysis experimentally.

5. EXPERIMENTAL VALIDATION

In this section, we experimentally evaluate the information leakage due to different deduplication techniques using real datasets.

5.1 Datasets

In our experiments, we rely on three real and a synthetic dataset, as seen in Table 1. Dataset 1 consists of 113 GiB of files stored in a shared repository of an academic institution. Dataset 2 corresponds to 13.4 TiB of data extracted from a subset of the publicly available collection of a previous study [2, 15]. Additionally, Dataset 3 was extracted from a file storage of an industrial research lab. Finally, Dataset 4 is a synthetic dataset consisting of employee contracts derived from the same template (cf. Section 4.2.1). For Datasets 2 and 3, we only had access to content hashes of the deduplicated objects (i.e., file hashes for file-based deduplication and block hashes for the block-based deduplication techniques). Notice that we purged empty files in all studied datasets as they cannot leak any content.

In our evaluation, Dataset 3 is used only for cross-validation to confirm our findings about the probability of file presence. Dataset 4 serves as the basis to evaluate the information leakage derived from template attacks (cf. Section 5.4).

Table 1 also lists the available block sizes for fixed-sized and CDC-based deduplication techniques for each dataset.

In Tables 4 and 5 in the Appendix, we evaluate the storage reduction due to the previously studied deduplication techniques for Datasets 1 and 2. When measuring storage reduction due to deduplication, we also account for the addi-

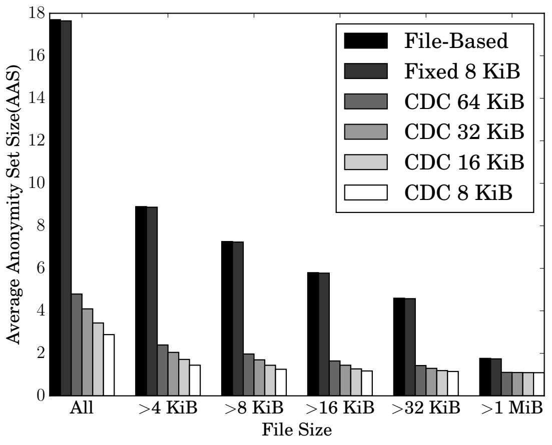


Figure 6: The sizes of anonymity sets plotted depending on minimum file sizes in Dataset 2.

	File Sizes	#Files	AAS	Unique
File-Based	All	219,578	3.403	18.9%
	≥ 2 KiB	150,214	2.404	27.6%
	≥ 1 MiB	7,704	1.204	78.6%
Fixed 4 KiB	All	219,578	3.402	18.9%
	≥ 2 KiB	150,214	2.403	27.6%
	≥ 1 MiB	7,704	1.200	78.9%
CDC 4 KiB	All	219,578	1.728	54.4%
	≥ 2 KiB	150,214	1.201	79.5%
	≥ 1 MiB	7,704	1.004	99.3%
CDC 8 KiB	All	219,578	2.142	41.8%
	≥ 4 KiB	120,437	1.224	91.5%
	≥ 1 MiB	7,704	1.004	99.3%
CDC 1 MiB	All	219,578	3.339	19.6%
	≥ .5 MiB	11,525	1.023	96.5%
	≥ 1 MiB	7,704	1.017	97.3%

Table 2: Average Anonymity Sets and Unique Deduplication Fingerprint Percentages for Dataset 1.

	File Sizes	#Files	AAS	Unique
File-Based	All	13,646,344	17.698	3.4%
	≥ 4 KiB	6,827,413	8.902	6.8%
	≥ 1 MiB	526,720	1.762	49.9%
Fixed 8 KiB	All	13,646,344	17.642	3.4%
	≥ 4 KiB	6,827,413	5.774	10.6%
	≥ 1 MiB	526,720	1.740	50.3%
CDC 8 KiB	All	13,646,344	2.881	3.23%
	≥ 4 KiB	6,827,413	1.442	64.5%
	≥ 1 MiB	526,720	1.089	88.4%
CDC 16 KiB	All	13,594,861	3.428	26.7%
	≥ 8 KiB	5,497,203	1.438	63.9%
	≥ 1 MiB	526,921	1.092	87.9%
CDC 32 KiB	All	13,609,646	4.092	21.9%
	≥ 16 KiB	4,341,011	1.440	62.8%
	≥ 1 MiB	526,755	1.097	87.3%
CDC 64 KiB	All	13,604,132	4.790	18.3%
	≥ 32 KiB	3,366,832	1.425	62.6%
	≥ 1 MiB	527,787	1.103	86.6%

Table 3: Average Anonymity Sets and Unique Deduplication Fingerprint Percentages for Dataset 2.

tional storage costs for storing file/block indexes. As mentioned earlier, the size of such indexes varies greatly with the different techniques as the number of stored objects is very different. Notice that there are a number of approaches for managing such indexes [16]; in this paper, we conservatively allocate 32 bytes per block hash, and 20 bytes per file path hash.

We observe that, for all studied datasets, CDC-based schemes using Rabin fingerprints with average block sizes of 4 KiB and 8 KiB exhibit the highest storage reductions (up to 20% in Dataset 1 and 70% in Dataset 2). File-based deduplication techniques achieve lower storage reduction when compared to block-based techniques (e.g., file-based deduplication reduces storage costs by 5% in Dataset 1 and by 50% in Dataset 2). We further observe that fixed-sized block deduplication techniques considerably improve storage costs when compared to their file-based counterparts; these techniques however result in lower savings when compared to CDC-based schemes. Our results show that storage savings of fixed-block deduplication techniques are comparable to the arithmetic mean of storage savings from CDC-based and file-based deduplication techniques.

5.2 Anonymity Sets

We now evaluate the size of the anonymity sets and the uniqueness of the deduplication fingerprints for the various studied deduplication techniques. As the size of the anonymity set heavily depends on the file size, we filter the datasets according to the file sizes. For each deduplication algorithm, we measure average anonymity set size (AAS) as well as the percentage of files that are uniquely identified by their deduplication fingerprint for: *(i)* the complete dataset, *(ii)* files with bigger size than half the average block size (where applicable), and *(iii)* files bigger than 1 MiB.

Our results are detailed in Tables 2 and 3, and summarized in Figure 6. Our results indicate that, in the case of file-based deduplication, larger files tend to have smaller anonymity sets and a higher percentage of unique deduplication fingerprints. Notice that fixed-sized deduplication techniques achieve almost identical anonymity sets (and percentages of unique fingerprints) to file-based techniques. Therefore, we list only one example for fixed-sized techniques in Tables 2 and 3. As outlined in section 4.3, the different anonymity set sizes of file-based and fixed-sized deduplication techniques are due to intra-file deduplication.

CDC-based schemes exhibit significantly smaller anonymity sets and higher percentages of unique fingerprints. In particular, for files bigger than 4 KiB, 91.5% and 64.5% of files in Datasets 1 and 2, respectively, are uniquely identified by their deduplication fingerprints. This is significantly more than the 34.4% and 6.8% (for Dataset 1 and 2, respectively) that are measured in the case of file-based deduplication and fixed-block-based deduplication. Notice that half the average block size corresponds to the minimum block size, and is thus the minimum threshold for performing deduplication.

We study the effect of the block sizes on the leakage in Figure 7. Here, we plot the AAS w.r.t. the size of files in

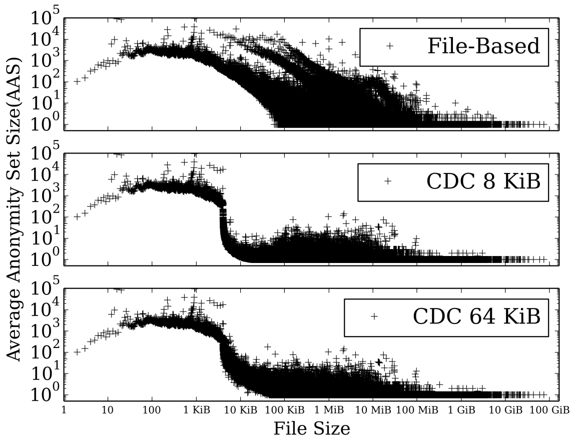


Figure 7: Anonymity set sizes plotted as a function of the file size. We compare three deduplication techniques over Dataset 2. Smaller anonymity sets result in higher leakage.

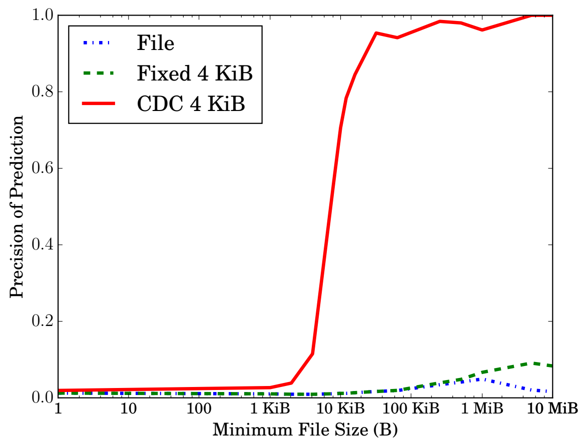


Figure 8: Cross-Dataset Validation of deduplication fingerprints as storage distinguishers. CDC-based deduplication fingerprints with 4 KiB blocks are best distinguisher.

Dataset 2.[2] While we see a clear difference between the file-based and the CDC-based AAS, the two CDC-based plots mostly differ for file sizes between 4K and 32K; for files smaller than this threshold, the algorithms translate to standard file-based deduplication algorithms (due to the minimum deduplication threshold).

Our experiments confirm our analysis in section 4.3. Namely, CDC-based deduplication techniques exhibit a significantly smaller anonymity sets and therefore higher leakage when compared to those of fixed-block and file-based algorithms. The latter algorithms result in similar information leakage since we assume that the adversary cannot observe any intra-file deduplication. We discuss the influence of intra-file deduplication in section 6.1.

5.3 Cross-Dataset Validation

We now evaluate information leakage due to deduplication *without knowledge of the storage domain*. To this end, we acquire all files from Dataset 1, compute their deduplication fingerprints and their content hashes and check them against the storage graph of Dataset 3. For each file out of Dataset 1 we record the number of candidates in Dataset 3, and determine whether the file was actually stored, i.e., whether the content hashes match the content hashes of a candidate. This allows us to measure the true and false positives associated with deduplication fingerprints.

In Figure 8, we plot the success rate of using different deduplication fingerprints as distinguisher, w.r.t. different minimal file sizes. The success rate is defined as the ratio of true positives over the number of candidates, i.e., as the percentage of candidates that are correctly identified. Using file-based deduplication fingerprints as distinguisher, yields a low success rate as all files with the same file size are possible candidates. Notice that the success rate for using fixed-sized block-based deduplication fingerprints as a distinguisher is similar to that of file-based deduplication fingerprints for small files; for large files the precision is slightly bigger due to intra-file deduplication.

For CDC-based deduplication fingerprints, we observe a higher success rate for larger files. In particular, for files above 32 KiB, we obtain a success rate above 90%. This means that if a candidate for such a file is found, then it is highly likely that the file is actually stored. This conforms with our analysis in Section 4.3.3. As shown in Figure 8, the success rate of the storage prediction significantly increases between 2 KiB and 10 KiB owing to the decrease of the collision probability of CDC-based deduplication fingerprints. However, we also observe that there are a few false positives for larger files. Namely, there are three collisions for files larger than 100 KiB. Such collisions would be extremely unlikely given our analysis in Section 4.3.3. Notice that this analysis assumes randomly chosen files of a certain size. We believe that this discrepancy in the number of collisions originates from the fact that those particular files were quite similar (e.g., different versions of the same library). However, we could not verify this assumption since we did not have access to the plaintexts of Dataset 3.

Overall, we conclude that even without knowledge of the storage domain, $\mathcal{A}$ can reliably infer whether large files are stored on $\mathcal{S}$, since deduplication fingerprints provide a good distinguisher for stored files. Furthermore, most false positives obtained for large files often result from the fact that these files are similar (e.g., identical files with different creation timestamps).

5.4 Template Attacks

In this section, we evaluate the impact of template attacks (cf. Section 4.2.1), in which the adversary possesses additional information about the storage domain.

Here, we evaluate the following scenario. A company stores employment contracts on the cloud storage. Each contract consists of a PDF of two pages comprising a fixed part and variable parts. The variable parts correspond to em-

[2] We chose Dataset 2 since it is the biggest available dataset.

ployee names, salaries, the contracts' start dates, and some internal PDF variables.

In our evaluation, we generated the employee names by choosing 50 randomly names out of the most common first and surnames within the United States[3]. We assume that the salary takes one of five possible values and that the start date of employees is public knowledge (e.g., can be extracted from social media). Finally, we assume that timestamps of PDF files are integer numbers (at one-second precision) [11]. We further assume that the PDFs were created on the start date of the contract between 8:00 am and 9:00 am, which amounts to a total of 3,600 possible values.

Given the variables and their ranges, we generate Dataset 4 with 900,000 files. These files have a joint size of 33 GiB. To evaluate the impact of template attacks for different deduplication techniques, we randomly chose one file per employee name. Given these 50 files, we compute the true positive rate for predicting the correct salary defined by the ratio of correctly predicted salaries over 50. We independently repeat this experiment ten times.

Our results indicate that CDC-based deduplication with an average block size of 4 KiB exhibits the highest true positive rate of 37.2%; here, we can predict 14.4% of the salaries with 100% probability. This leakage is reduced by almost 30% when block sizes of 8 KiB or 1 MiB are used. Since Dataset 4 exhibited no intra-file deduplication, all fixed-sized deduplication techniques exhibited the same leakage as file-based deduplication. Namely, salaries were predicted with a true positive rate of 30.08% and 7.8% of the salaries were predicted with 100% probability.

Overall, we find that template attacks can result in a significant privacy leakage. Although such attacks require knowledge of the "template", we argue that templates are often public or can be easily extracted by observing public files.

6. DISCUSSION AND POSSIBLE COUNTERMEASURES

In this section, we discuss further insights with respect to our findings and we explore the solution space to prevent information leakage associated with existing data deduplication techniques.

6.1 Impact of Server-Side Deduplication

We start by analyzing the impact of server-side deduplication on our findings. Recall that when server-side deduplication is used, the adversary acquires additional information with respect to intra-file deduplication (i.e., the adversary learns whether a given file contains duplicate blocks). In case of client-side deduplication, duplicate blocks within the same file are not uploaded onto the storage—effectively hiding the size of the uploaded file. As such, intra-file deduplication increases the difficulty for an adversary to correctly estimate the size of the anonymity set if the file storage domain is not known beforehand. This also suggests that information leakage is only expected to increase when server-side deduplication is used.

As shown in Table 6 in the appendix, intra-file deduplication is more likely to occur in larger files. In particular, in Dataset 2, 69.11% of files greater than 10 MiB exhibit considerable intra-file deduplication. Additionally, deduplication schemes with smaller block sizes result in larger intra-file deduplication rates. This is true for both fixed-sized block-based and CDC-based schemes.

6.2 Querying for a Set of Files

So far, we have studied storage inference for single target files. However, we argue that information leakage can be further exacerbated if the adversary $\mathcal{A}$ wants to determine whether a set of files $\mathcal{F}$ is *entirely* stored on $\mathcal{S}$. In this case, the Probability of Storage from Equation1 changes to:

$$\begin{aligned}\mathrm{PoS}(\mathcal{F}, G_o, D_o) &= P(\mathcal{F} \subseteq X \mid G = G_o, D = D_o) \\ &= \frac{\sum\limits_{X_p : \mathcal{F} \subseteq X_p} P(X = X_p, G = G_o, D = D_o)}{\sum\limits_{X_p} P(X = X_p, G = G_o, D = D_o)}\end{aligned}$$

Here, if $\mathcal{A}$ cannot compute the Probability of Storage, due to incomplete knowledge of the storage domain, $\mathcal{A}$ still benefits from querying for a whole set of files at once. Firstly, $\mathcal{A}$ generates a probe graph $\mathcal{P}$, a storage graph containing all files of $\mathcal{F}$. Notice that the probe graph is not just the sum of all the trees. If the files in $\mathcal{F}$ have shared objects, the respective trees will be connected. Subsequently, $\mathcal{A}$ finds the candidate set of $\mathcal{P}$ in G, i.e., $\mathcal{A}$ finds all isomorphisms between $\mathcal{P}$ and a subgraph of G. We argue that the shared objects (and the parallel query for all files) reduce the candidate set of $\mathcal{P}$ and provide a more reliable prediction whether $\mathcal{F}$ is *entirely* stored on $\mathcal{S}$ when compared to the case where the adversary queries for individual files.

6.3 Leakage through Frequency Analysis

So far, our analysis has focused solely on information acquired from the object size and timestamps. However, we point out that the adversary can additionally infer how often and in which contexts deduplicated objects are uploaded/retrieved. For block-based deduplication techniques, $\mathcal{A}$ can infer the number of files an object appears in. $\mathcal{A}$ can perform such "frequency analysis" by measuring the node degree of data nodes in the storage graph. Clearly, $\mathcal{A}$ can combine the use of frequency analysis techniques with our aforementioned techniques in order to infer additional information with respect to the stored files.

For example, the "zero-block" is often the most frequently deduplicated object [15]. In our setting, $\mathcal{A}$ can identify the data node in the storage graph that refers to the zero-block through frequency analysis. $\mathcal{A}$ can label the zero-block in the storage graph and the trees of the target files. These labels restrict possible isomorphisms, reduce candidate sets, and make our attack even more precise. $\mathcal{A}$ can additionally fingerprint other common data nodes such as recurring file headers or footers if $\mathcal{A}$ can reliably infer them by means of similar frequency analysis.

[3]Data extracted from https://www.ssa.gov/OACT/babynames/ and http://www2.census.gov/topics/genealogy/2000surnames/Top1000.xls

6.4 Minimizing Leakage in Deduplicated Storage Systems

In the previous section, we have shown that existing deduplication schemes leak considerable information about the encrypted stored files. Namely, existing public chunking algorithms allow an adversary to construct deduplication fingerprints and compare the simulated deduplication fingerprints to those on the storage.

One possible way to prevent such leakage would be pad objects with additional data in an attempt to effectively obfuscate their size. However, such a solution would only increase the storage overhead and effectively strip away the benefits of data deduplication.

In order to truly hide the information leaked by deduplication fingerprints, clients could use a non-predictable secret chunking function that exhibits sufficient entropy, e.g., AES with a secret key. Clearly, this key has to be shared securely accross all clients—which is a challenging task. A possible solution to realize this would be for clients to perform content-based chunking by executing secure multiparty computation (MPC) with the storage server. This would clearly come at the expense of performance, since the fastest known AES variant implementation of MPC requires around 3.9 ms per per 128-bit block [13]. This amounts to 86 hours of computation per GiB of deduplicated data. Alternatively, the chunking key could be protected using trusted computing. For example, the key could be pre-deployed on a tamper-resistant smart card, or on a TPM chip.

Notice that our findings suggest that fixed-sized block-based deduplication schemes offer a strong trade-off between storage savings (cf. Section 5.1) and information leakage (cf. Section 5.2).

7. RELATED WORK

Data deduplication has received considerable research attention [16, 21].

Harnik *et al.* [10] describe a number of threats posed by client-side data deduplication, in which an adversary can learn if a file is already stored in a particular cloud by guessing the hashes of predictable messages. This information leakage can be eliminated through the reliance on Proofs of Ownership schemes (PoW) [5, 9], which require a client to prove full possession of the file.

Douceur *et al.* [6] introduced convergent encryption, where the encryption key is derived from the plaintext to ensure that identical plaintexts result in identical ciphertexts. This allows clients that do not share any keying material to benefit from cross-user deduplication. However, convergent encryption is not semantically secure [4] and only offers confidentiality for unpredictable file contents. To remedy this, Bellare *et al.* [3] proposed a server-side deduplication scheme, DupLESS, which employs a key server to generate file-specific keys and resist brute-force key search attacks. Stanek *et al.* [20] presented an encryption scheme where unpopular data is protected with semantic security while popular data is deduplicated effectively using convergent encryption. Once files become popular, they undergo a seamless transition. Similarly, Puzio *et al.* [18] proposed PerfectDedup, a scheme that detects the popularity of data blocks using perfect hashing. PerfectDedup protects data confidentiality by encrypting unpopular data semantically secure and deduplicates popular data using convergent encryption. Armknecht *et al.* [1] presented ClearBox, a system that performs transparent client-side deduplication and allows clients to pay according to their actual storage consumption.

Puzio *et al.* [17] studied how to combine block-level deduplication and data confidentiality. The authors proposes ClouDedup which is based on convergent encryption and a proxy server that adds an additional layer of encryption. The resulting ciphertexts are then stored at the storage provider.

8. CONCLUDING REMARKS

Most existing cloud providers reduce their storage costs by leveraging data deduplication when storing clients' data. Namely, existing cloud solutions store duplicate data uploaded by different users only once—thus significantly saving storage costs.

In this paper, we analyzed the information leaked to a curious storage server in systems that rely on client-side deduplication and encryption. We showed that even if the encryption key is unknown to the storage server, the latter can still acquire considerable information about the stored files. Our results show that data deduplication offers a strong distinguisher for a curious storage server in guessing which files are stored. We also showed that the information leakage associated with deduplication is further exacerbated when CDC-based deduplication techniques are used. We confirmed our analysis by means of thorough experiments using four different datasets.

Finally, our analysis shows that there are no bullet-proof solutions to deter information leakage associated with the use of data deduplication. We argue that the reliance on fixed-sized block-based deduplication techniques emerges as a strong trade-off between storage efficiency and privacy leakage. We therefore hope that our findings increase the awareness of cloud providers and users with respect the privacy risks associated with data deduplication.

Acknowledgements

This work was partly supported by the TREDISEC project (G.A. no 644412), funded by the European Union (EU) under the Information and Communication Technologies (ICT) theme of the Horizon 2020 (H2020) research and innovation programme and by the Zurich Information Security & Privacy Center. This paper represents only the views of the authors. The authors would like to thank the anonymous reviewers for their detailed comments. Additionally, the authors would like to thank Damian Gruber, Wenting Li, Mario Lučić, Lara Schmid, and Alessandro Sforzin for the helpful comments and discussions.

9. REFERENCES

[1] Frederik Armknecht, Jens-Matthias Bohli, Ghassan O. Karame, and Franck Youssef. Transparent data

deduplication in the cloud. CCS '15, pages 886–900, New York, NY, USA, 2015. ACM.

[2] Storage Networking Industry Association. SNIA IOTTA Repository. http://iotta.snia.org/traces/3382.

[3] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Dupless: Server-aided encryption for deduplicated storage. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 179–194. USENIX Association, 2013.

[4] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Message-locked encryption and secure deduplication. EUROCRYPT '13. Springer, 2013.

[5] Roberto Di Pietro and Alessandro Sorniotti. Boosting efficiency and security in proof of ownership for deduplication. ASIACCS '12, pages 81–82, New York, NY, USA, 2012. ACM.

[6] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *ICDCS*, pages 617–624, 2002.

[7] Kave Eshghi and Hsiu K. Tang. A framework for analyzing and improving content-based chunking algorithms. Technical report, 2005. http://www.hpl.hp.com/techreports/2005/HPL-2005-30R1.html.

[8] William Feller. *An Introduction to Probability Theory and Its Applications*. Wiley, 2nd edition, 1971.

[9] Shai Halevi, Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Proofs of ownership in remote storage systems. CCS '11, pages 491–500, New York, NY, USA, 2011. ACM.

[10] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, 2010.

[11] Adobe Systems Incorporated. Document management - Portable document format - Part 1: PDF 1.7, 2008. https://wwwimages2.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/PDF32000_2008.pdf.

[12] Kevin B. Korb and Ann E. Nicholson. *Bayesian Artificial Intelligence*. CRC Press, second edition, 2010.

[13] Sven Laur, Riivo Talviste, and Jan Willemson. From oblivious aes to efficient and secure database join in the multiparty setting. ACNS'13. Springer-Verlag, 2013.

[14] Dirk Meister and André Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. ACM, 2009.

[15] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. *Trans. Storage*, 7(4):14:1–14:20, February 2012.

[16] João Paulo and José Pereira. A survey and classification of storage deduplication systems. *ACM Comput. Surv.*, 47(1):11:1–11:30, June 2014.

[17] Pasquale Puzio, Refik Molva, Melek Önen, and Sergio Loureiro. Block-level de-duplication with encrypted data. *Open Journal of Cloud Computing (OJCC)*, 1(1):10–18, 2014.

[18] Pasquale Puzio, Refik Molva, Melek Önen, and Sergio Loureiro. *PerfectDedup: Secure Data Deduplication*, pages 150–166. Springer International Publishing, Cham, 2016.

[19] Michael O. Rabin. Fingerprinting by random polynomials. *Harvard Aiken Computation Laboratory*, pages 1–12, 1981.

[20] Jan Stanek, Alessandro Sorniotti, Elli Androulaki, and Lukas Kencl. A Secure Data Deduplication Scheme for Cloud Storage. In *18th International Conference on Financial Cryptography and Data Security (FC)*, 2014.

[21] Mark W. Storer, Kevin Greenan, Darrell D.E. Long, and Ethan L. Miller. Secure data deduplication. StorageSS '08, 2008.

[22] Marc van Leeuwen. Extended stars-and-bars problem, 2013. http://math.stackexchange.com/a/554237.

APPENDIX

Dataset 1	Storage Savings	Object Savings	#Objects	Object Size	Index Size
File	5.3%	24.0%	219,578	133.8 GiB	25.0 MiB
Fixed 1 MiB	9.6%	19.2%	338,994	127.7 GiB	35.6 MiB
Fixed 4 KiB	12.2%	13.5%	31,997,908	121.6 GiB	2.5 GiB
Fixed 8 KiB	12.4%	12.8%	16,201,377	122.6 GiB	1.3 GiB
CDC 1 MiB	11.0%	19.9%	328,609	125.8 GiB	34.9 MiB
CDC 4 KiB	20.1%	21.6%	27,842,978	110.6 GiB	2.3 GiB
CDC 8 KiB	19.6%	20.3%	14,210,609	112.5 GiB	1.2 GiB

Table 4: Storage Savings due to different deduplication techniques in Dataset 1.

Dataset 2	Storage Savings	Object Savings	#Objects	Object Size	Index Size
File	52.4%	74.1%	13,646,344	6.4 TiB	3.0 GiB
Fixed 8 KiB	60.8%	61.7%	701,568,394	5.2 TiB	79.1 GiB
Fixed 16 KiB	60.0%	61.0%	365,965,683	5.3 TiB	41.2 GiB
Fixed 32 KiB	59.0%	60.8%	193,653,052	5.5 TiB	22.2 GiB
Fixed 64 KiB	58.2%	61.1%	104,375,784	5.6 TiB	12.5 GiB
CDC 8 KiB	70.6%	71.6%	535,997,134	3.9 TiB	75.3 GiB
CDC 16 KiB	69.7%	71.1%	333,162,476	4.0 TiB	46.5 GiB
CDC 32 KiB	68.8%	70.4%	231,535,026	4.2 TiB	32.0 GiB
CDC 64 KiB	67.8%	69.8%	181,628,779	4.3 TiB	25.0 GiB

Table 5: Storage Savings due to different deduplication techniques in Dataset 2.

File Sizes	All	≥1 MiB	≥10 MiB	≥100 MiB	≥1 GiB
#Files	13,594,861	526,720	66,224	4,508	535
Fixed 8 KiB	1.42%	18.98%	43.79%	46.27%	75.70%
Fixed 16 KiB	0.90%	13.48%	33.41%	38.07%	66.92%
Fixed 64 KiB	0.28%	5.29%	8.74%	26.15%	59.93%
CDC 8 KiB	2.81%	41.37%	69.11%	67.30%	84.89%
CDC 16 KiB	2.42%	35.93%	65.80%	65.96%	83.98%
CDC 64 KiB	2.02%	29.65%	58.32%	60.58%	83.52%

Table 6: Intra-File Deduplication for Dataset 2. For each configuration we give the percentage of files in which intra-file deduplication occurs.