Information Leakage in Encrypted Deduplication via Frequency Analysis: Attacks and Defenses[*]

Jingwei Li[1], Patrick P. C. Lee[2], Chufeng Tan[1], Chuan Qin[2], and Xiaosong Zhang[1]

[1]University of Electronic Science and Technology of China

[2]The Chinese University of Hong Kong

Technical Report

Abstract

Encrypted deduplication combines encryption and deduplication to simultaneously achieve both data security and storage efficiency. State-of-the-art encrypted deduplication systems mainly build on deterministic encryption to preserve deduplication effectiveness. However, such deterministic encryption reveals the underlying frequency distribution of the original plaintext chunks. This allows an adversary to launch frequency analysis against the ciphertext chunks and infer the content of the original plaintext chunks. In this paper, we study how frequency analysis affects information leakage in encrypted deduplication, from both attack and defense perspectives. Specifically, we target backup workloads, and propose a new inference attack that exploits *chunk locality* to increase the coverage of inferred chunks. We further combine the new inference attack with the knowledge of chunk sizes and show its attack effectiveness against variable-size chunks. We conduct trace-driven evaluation on both real-world and synthetic datasets and show that our proposed attacks infer a significant fraction of plaintext chunks under backup workloads. To defend against frequency analysis, we present two defense approaches, namely MinHash encryption and scrambling. Our trace-driven evaluation shows that our combined MinHash encryption and scrambling scheme effectively mitigates the severity of the inference attacks, while maintaining high storage efficiency and incurring limited metadata access overhead.

1 Introduction

To manage massive amounts of data in the wild, modern storage systems employ *deduplication* (see Section 2.1) to eliminate content duplicates and save storage space. The common deduplication approach is to store only data copies, called *chunks*, that have unique content among all already stored chunks. Field studies have demonstrated that deduplication achieves significant storage savings in production, for example, by 50% in primary storage [49] and up to 98% in backup storage [62]. Deduplication is also adopted by commercial cloud storage services (e.g., Dropbox, Google Drive, Bitcasa, etc.) for cost-efficient outsourced data management [43].

In the security context, combining encryption and deduplication, referred to as *encrypted deduplication* (see Section 2.2), is essential for protecting against information leakage in deduplicated storage. Conventional (symmetric) encryption requires that users encrypt data with their own distinct secret keys. As a result, duplicate plaintext chunks will be encrypted into distinct ciphertext chunks, thereby prohibiting deduplication across different users. To preserve deduplication effectiveness, encrypted deduplication requires that each chunk be encrypted with a secret key derived from the chunk content itself, so that identical plaintext

---

[*]An earlier conference version of this paper appeared at the 47th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2017) [42]. In this extended version, we propose new attack and defense schemes, include a new dataset in our evaluation, and add new prototype experiments.
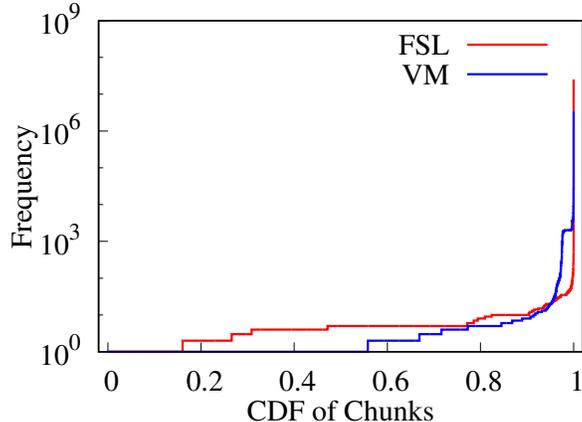
Figure 1: Frequency distributions of chunks with duplicate content in the FSL and VM datasets (see Section 5 for dataset details).

chunks are always encrypted into identical ciphertext chunks for deduplication. Bellare *et al.* [13] propose a cryptographic primitive called *Message-locked encryption (MLE)* to formalize the key derivation requirement of encrypted deduplication, in which an MLE scheme consists of a key generation algorithm that maps the content of a message (or chunk in our case) into a secret key for symmetric encryption/decryption; in particular, convergent encryption [22] is one classical instantiation of MLE by deriving the secret key through the hash of a chunk. On top of MLE, several storage systems address additional security issues, such as brute-force attacks [12], key management failures [24], side-channel attacks [31], and access control [53].

However, we argue that existing MLE schemes cannot fully protect against information leakage, mainly because their encryption approaches are *deterministic*. That is, each ciphertext chunk is encrypted by a key that is deterministically derived from the original plaintext chunk. Thus, an adversary, which can be malicious users or storage system administrators, can analyze the frequency distribution of ciphertext chunks and infer the original plaintext chunks based on *frequency analysis*. We observe that practical deduplicated storage workloads often exhibit skewed frequency distributions in terms of the occurrences of chunks with the same content. Figure 1 justifies our observation, by depicting the skewed frequency distributions of chunks in the real-world FSL and VM datasets (see Section 5 for the dataset details). For example, the FSL dataset has 99.8% of chunks occurring fewer than 100 times, while around 30 out of 41 million chunks (or 0.00007% of chunks) occur over 10,000 times; the VM dataset has 97% of chunks occurring fewer than 100 times, while around 15,000 out of 35 million chunks (or 0.04% of chunks) occur over 10,000 times. Such skewed frequency distributions allow the adversary to accurately differentiate chunks by their frequencies via frequency analysis. On the other hand, while frequency analysis is a historically well-known cryptanalysis attack [4], the practical implications of frequency analysis against encrypted deduplication remain unexplored.

In this paper, we conduct an in-depth study of how frequency analysis practically affects information leakage in encrypted deduplication. Our study spans both attack and defense perspectives, and is specifically driven by the characteristics of storage workloads in deduplication systems.

On the attack side, we propose a new inference attack called the *locality-based* attack, which extends classical frequency analysis to accurately infer ciphertext-plaintext chunk pairs in encrypted deduplication. The main novelty of the locality-based attack is to exploit *chunk locality*, a common property in practical backup workloads. Chunk locality states that chunks are likely to re-occur together with their neighboring chunks across different versions of backups, mainly because in practice, changes to backups often appear in few clustered regions of chunks, while the remaining regions of chunks will appear in the same order in previous backups. Previous studies have exploited chunk locality to improve deduplication performance

and mitigate indexing overhead (e.g., [45, 65, 67]). Here, we adapt this idea from a security perspective into frequency analysis: if a plaintext chunk M corresponds to a ciphertext chunk C, then the neighboring plaintext chunks of M are likely to correspond to the neighboring ciphertext chunks of C.

Our trace-driven evaluation, using both real-world and synthetic datasets, shows that the locality-based attack can infer significantly more ciphertext-plaintext pairs than classical frequency analysis. For example, for the real-world FSL dataset, the locality-based attack can infer up to 23.2% of the latest backup data, while the basic attack that directly applies classical frequency analysis can only infer 0.0001% of data. If a limited fraction (e.g., 0.2%) of plaintext information of the latest backup is leaked, the inference rate of the locality-based attack can increase up to 27.5%.

We further combine the locality-based attack with the knowledge of chunk sizes, and propose an advanced locality-based attack against variable-size chunks. The advanced locality-based attack maps ciphertext chunks to some plaintext chunks with similar sizes, and further increases the inference rate.

Our inference attacks are harmful in practice, even though the underlying symmetric encryption remains secure. One security implication of our inference attacks is that they can identify critical chunks in an encrypted backup snapshot. Given the plaintext chunks of some critical files (e.g., password files) in an old backup, an adversary can infer the ciphertext chunks in the latest backup corresponding to those critical plaintext chunks. It can then launch specific attacks against such identified ciphertext chunks; for example, by dedicatedly corrupting (e.g., deleting or modifying) such ciphertext chunks, the adversary can make the underlying critical plaintext information unrecoverable.

On the defense side, we present two defense approaches to combat the inference attacks. The first one is *MinHash encryption*, which derives a common encryption key based on a set of adjacent chunks, such that some identical plaintext chunks can be encrypted into multiple distinct ciphertext chunks. Note that MinHash encryption has been shown to effectively reduce the overhead of server-aided MLE [53]; here we show how it can also be used to break the deterministic nature of encrypted deduplication and disturb the frequency ranking of ciphertext chunks. The second one is *scrambling*, which randomly shuffles the original chunk ordering during the deduplication process in order to break chunk locality. Our trace-driven evaluation shows that the combined MinHash encryption and scrambling scheme can suppress the inference rate to only 0.23% for the FSL dataset.

We also evaluate the storage efficiency and deduplication performance of the combined MinHash encryption and scrambling scheme. First, the combined scheme maintains the high storage saving achieved by deduplication, and its storage saving is only up to 3.6% less than that of the original MLE, which uses chunk-based deduplication. In addition, we build a realistic deduplication prototype based on DDFS [67] and evaluate the on-disk metadata access overhead. We show that the combined scheme incurs up to 1.2% additional metadata access overhead compared to the original MLE, and it incurs even less metadata access overhead when there is sufficient memory for metadata caching. Our findings suggest that the combined scheme adds limited overhead to both storage efficiency and deduplication performance in practical deployment, while effectively defending against frequency analysis.

The source code of our attack and defense implementations as well as the deduplication prototype is available at: **http://adslab.cse.cuhk.edu.hk/software/freqanalysis**.

The remainder of the paper proceeds as follows. Section 2 reviews the basics of encrypted deduplication and frequency analysis. Section 3 defines the threat model. Section 4 presents our proposed inference attacks based on frequency analysis. Section 5 presents the evaluation results of our proposed inference attacks. Section 6 presents the defense schemes against the inference attacks. Section 7 presents the evaluation results of our defense schemes. Section 8 reviews the related work, and finally, Section 9 concludes the paper.

2 Basics

2.1 Deduplication

Deduplication can be viewed as a coarse-grained compression technique to save storage space. We focus on chunk-based deduplication that operates at the granularities of chunks. Specifically, a deduplication system partitions input data into variable-size chunks through content-defined chunking (e.g., Rabin fingerprinting [54]), which identifies chunk boundaries that match specific content patterns so as to remain robust against content shifts [25]. We can configure the minimum, average, and maximum chunk sizes in content-defined chunking for different granularities. After chunking, each chunk is identified by a *fingerprint*, which is computed from the cryptographic hash of the content of the chunk. Any two chunks are said to be identical if they have the same fingerprint, and the collision probability that two non-identical chunks have the same fingerprint is practically negligible [16]. Deduplication requires that only one physical copy of identical chunks is kept in the storage system, while any identical chunk refers to the physical chunk via a small-size reference.

To check if any identical chunk exists, the deduplication system maintains a *fingerprint index*, a key-value store that holds the mappings of all fingerprints to the addresses of physical chunks that are currently stored. For each file, the storage system also stores a *file recipe* that lists the references to all chunks of the file for future reconstruction.

2.2 Encrypted Deduplication

Encrypted deduplication ensures that all physical chunks are encrypted for confidentiality (i.e., data remains secret from unauthorized users and even storage system administrators), while the ciphertext chunks that are originated from identical plaintext chunks can still be deduplicated for storage savings. As stated in Section 1, message-locked encryption (MLE) [13] is a formal cryptographic primitive for encrypted deduplication, in which each chunk is encrypted/decrypted by a secret key that is derived from the chunk content itself through some key generation algorithm. For example, convergent encryption [22] is one popular MLE instantiation, and uses the cryptographic hash of a chunk as the corresponding symmetric key. This ensures that identical plaintext chunks must be encrypted into the identical ciphertext chunks, thereby preserving deduplication effectiveness. Note that the encrypted deduplication system needs to maintain a *key recipe* for each user to track the per-chunk keys for future decryption. Each key recipe is encrypted by the user's own secret key via conventional encryption for protection (see Section 3).

MLE is inherently vulnerable to the offline brute-force attack [13], which allows an adversary to determine which plaintext chunk is encrypted into an input ciphertext chunk. The brute-force attack works as follows. Suppose that the adversary knows the set of chunks from which the underlying plaintext chunk is drawn. Then for each chunk from the set, the adversary finds the chunk-derived key (whose key derivation algorithm is supposed to be publicly available), encrypts the chunk with the chunk-derived key, and finally checks if the output ciphertext chunk is identical to the input ciphertext chunk. If so, the plaintext chunk is the answer. Thus, MLE can only achieve security for *unpredictable* chunks [13], meaning that the size of the set of chunks is sufficiently large, such that the brute-force attack becomes infeasible.

To protect against the brute-force attack, DupLESS [12] realizes *server-aided MLE*, which outsources MLE key management to a dedicated *key manager* that is only accessible by authenticated clients. Each authenticated client needs to first query the key manager for the chunk-derived key. Then the key manager computes and returns the key via a deterministic key derivation algorithm that takes the inputs of both the chunk fingerprint and a system-wide secret maintained by the key manager itself. This makes the resulting ciphertext chunks appear to be encrypted by random keys from the adversary's point of view. In addition, the key manager limits the rate of key generation to slow down any online brute-force attack for querying
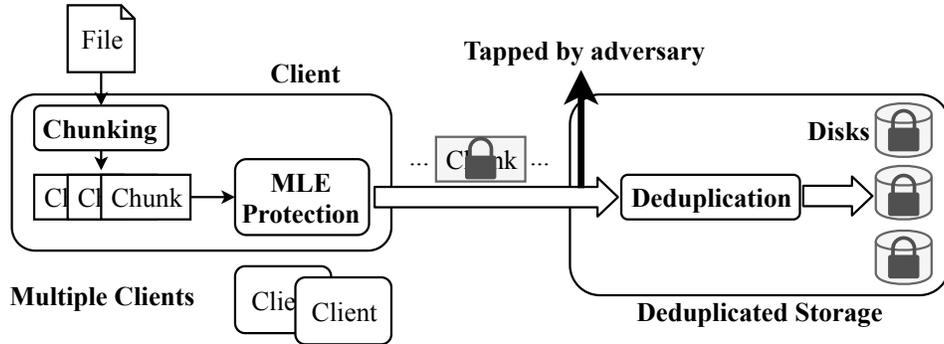
Figure 2: Architecture of encrypted deduplication.

the encryption keys. If the key manager is secure against adversaries, server-aided MLE ensures security even for predictable chunks; otherwise, it still maintains security for unpredictable chunks as in original MLE [13].

Most existing MLE implementations, either based on convergent encryption or server-aided MLE, follow *deterministic encryption* to ensure that identical plaintext chunks always form identical ciphertext chunks to make deduplication possible. Thus, they are inherently vulnerable to frequency analysis as shown in this paper. Some encrypted deduplication designs are based on non-deterministic encryption [3,11,13,46], yet they still keep deterministic components [13], incur high performance overhead [46], or require cryptographic primitives that are not readily implemented [3,11]. We elaborate the details in Section 8.

3 Threat Model

3.1 Overview

We focus on backup workloads, which have substantial content redundancy and are proven to be effective for deduplication in practice [62,67]. Backups are copies of primary data (e.g., application states, file systems, and virtual disk images) over time. They are typically represented as weekly full backups (i.e., complete copies of data) followed by daily incremental backups (i.e., changes of data since the last full backup), while the recent trend shows that full backups are now more frequently performed (e.g., every few days) in production [6]. Our threat model focuses on comparing different versions of full backups from the same primary data source at different times. In the following discussion, we simply refer to "full backups" as "backups".

Figure 2 shows the encrypted deduplication architecture considered in the paper. Suppose that multiple clients connect to a shared deduplicated storage system for data backups. Given an input file, a client divides file data into plaintext chunks that will be encrypted via MLE. It then uploads the ciphertext chunks to deduplicated storage. An adversary can eavesdrop the ciphertext chunks before deduplication and launch frequency analysis. We assume that the adversary is honest-but-curious, meaning that it does not change the prescribed storage system protocols or modify any stored data.

3.2 Auxiliary Information

To launch frequency analysis, the adversary should have access to the *auxiliary information* that provides ground truths about the backups being stored. Prior studies have proposed different approaches to obtain the auxiliary information to launch inference attacks. We briefly discuss several representative ones that inspire our work.

5

- Naveed *et al.* [50] examine inference attacks against the encrypted databases for electronic medical records, some of which are protected by deterministic encryption. To evaluate the feasibility of launching inference attacks, the authors obtain the auxiliary information from a public user dataset released by the government health services.

- Grubbs *et al.* [28] infer the plaintexts of the attributes of customer records (e.g., first name, last name, ZIP codes, birth dates, etc.) stored in an encrypted database. They obtain the auxiliary information regarding the plaintext distribution via the public census and survey datasets.

- Bindschaedler *et al.* [15] also infer the plaintexts of the attributes of encrypted customer records like Grubbs *et al.* [28], but use the public and purchased U.S. voter registration lists as the auxiliary information. The authors also use the older versions of purchased hospital-discharge data and public censor data to infer the newer versions of respective data.

- Grubbs *et al.* [27] focus on Ubuntu Internet Relay Chat (IRC) logs [2, 61] and extract the log keywords. They generate the keyword query distribution from one year's Ubuntu IRC logs as the auxiliary information to infer the encrypted keywords in the logs of a later year.

- Pouliot *et al.* [52] consider the inference attacks against the keywords in the Enron email dataset [37]. They first partition each user's emails into two non-overlapping sets (i.e., training and testing sets). They then generate the necessary auxiliary information from the training set, and infer the content of the testing set that is encrypted.

- Other studies [19, 33, 66] also leverage the Enron email dataset. They create a Zipfian synthetic keyword query distribution from the keyword list of the whole Enron dataset as the auxiliary information, and use it to infer the keywords in the original dataset that is encrypted.

We observe that previous studies mainly obtain the auxiliary information from *private* [15, 19, 27, 33, 52, 66] or *public* [15, 28, 50] sources. By private, we mean that the auxiliary information is originally protected but is obtained through unintended data releases [10], data breaches [29], or stolen storage devices [32]. By public, we mean that the auxiliary information can be legitimately accessed by the adversary.

In this work, we mainly focus on the private auxiliary information, which we model as the *plaintext chunks of a prior (non-latest) backup*. Nevertheless, in our evaluation (see Section 5), we also address the public auxiliary information, in which we choose a virtual disk image that is publicly accessible.

We note that the success of frequency analysis heavily depends on how accurate the available auxiliary information describes the backups. Our focus is *not* to address how to obtain accurate auxiliary information, which we pose as future work; instead, given the available auxiliary information, we study how an adversary can design severe attacks based on frequency analysis and how we can defend against the attacks.

3.3 Attack Modes

Based on the available auxiliary information (which describes a prior backup), the primary goal of the adversary is to *infer* the content of the plaintext chunks that are mapped to the ciphertext chunks of the latest backup. The attack can be based on two modes:

- *Ciphertext-only mode:* It models a typical case in which the adversary can access the ciphertext chunks of the latest backup (as well as the auxiliary information about a prior backup).

- *Known-plaintext mode:* It models a more severe case in which a powerful adversary not only can access the ciphertext chunks of the latest backup and the auxiliary information about a prior backup as in ciphertext-only mode, but also knows a small fraction of the ciphertext-plaintext chunk pairs about the latest backup (e.g., from stolen devices [21]).

6

In both attack modes, we make the following assumptions on the capabilities of an adversary.

- The adversary can monitor the processing sequence of the storage system and access the *logical order* of ciphertext chunks of the latest backup before deduplication. Our rationale is that existing deduplicated storage systems [65, 67] often process chunks in logical order, so as to effectively cache metadata for efficient deduplication.

- The adversary cannot access any metadata information (e.g., the fingerprint index, file recipes, key recipes of all files). In practice, we do not apply deduplication to the metadata, which can be protected by conventional encryption. For example, the file recipes and key recipes can be encrypted by user-specific secret keys.

- The adversary cannot identify which prior backup to which a stored ciphertext chunk belongs by analyzing the physical storage space, as the storage system can store ciphertext chunks in randomized physical addresses or in commercial public clouds (the latter is more difficult to access directly).

3.4 Other Attacks

While this work focuses on frequency analysis, another inference attack based on combinatorial optimization, called l_p-optimization, has been proposed to attack deterministic encryption [50]. Nevertheless, frequency analysis is shown to be as effective as the l_p-optimization attack [50], and later studies [40, 52] also state that both frequency analysis and l_p-optimization may have equivalent severity.

We do not consider other threats against encrypted deduplication, as they can be addressed independently by existing approaches. For example, the side-channel attack against encrypted deduplication [30, 31] can be addressed by server-side deduplication [31, 43] and proof of ownership [30]; the leakage of access pattern [33] can be addressed by oblivious RAM [57] and blind storage [51].

4 Attacks

We present inference attacks based on frequency analysis against encrypted deduplication. We first present a basic attack, which builds on classical frequency analysis to infer plaintext content in encrypted deduplication. We next propose a more severe locality-based attack, which enhances the basic attack by exploiting chunk locality. Furthermore, we combine the locality-based attack with the chunk size information, and propose an advanced locality-based attack against variable-size chunks.

Table 1 summarizes the major notation used in this paper. We first formalize the adversarial goal of our proposed attacks based on the threat model in Section 3. Let $\mathbf{C} = \langle C_1, C_2, \ldots \rangle$ be the sequence of ciphertext chunks in logical order for the latest backup, and $\mathbf{M} = \langle M_1, M_2, \ldots \rangle$ be the sequence of plaintext chunks in logical order for a prior backup (i.e., $\mathbf{M}$ is the auxiliary information). Both $\mathbf{C}$ and $\mathbf{M}$ show the logical orders of chunks before deduplication as perceived by the adversary (i.e., identical chunks may repeat), and each of them can have multiple identical chunks that have the same content. Note that both $\mathbf{C}$ and $\mathbf{M}$ do not necessarily have the same number of chunks. Furthermore, the i-th plaintext chunk M_i in $\mathbf{M}$ (where $i \geq 1$) is not necessarily mapped to the i-th ciphertext chunk in $\mathbf{C}$; in fact, M_i may not be mapped to any ciphertext chunk in $\mathbf{C}$, for example, when M_i has been updated before the latest backup is generated. Given $\mathbf{C}$ and $\mathbf{M}$, the goal of an adversary is to infer the content of the original plaintext chunks in $\mathbf{C}$.

We quantify the severity of an attack using the *inference rate*, defined as the ratio of the number of unique ciphertext chunks whose plaintext chunks are successfully inferred over the total number of unique ciphertext chunks in the latest backup; a higher inference rate implies that the attack is more severe.

7

Table 1: Major notation used in this paper.

Notation	Description
Defined in Section 4	
$\mathbf{C}$	sequence of ciphertext chunks $\langle C_1, \ldots \rangle$ in logical order for the latest backup
$\mathbf{M}$	sequence of plaintext chunks $\langle M_1, \ldots \rangle$ in logical order for a prior backup
$\mathbf{F_C}$	associative array that maps each ciphertext chunk in $\mathbf{C}$ to its frequency
$\mathbf{F_M}$	associative array that maps each plaintext chunk in $\mathbf{M}$ to its frequency
$\mathcal{T}$	set of inferred ciphertext-plaintext chunk pairs
$\mathcal{L}_C$	set of left neighbors of ciphertext chunk C
$\mathcal{L}_M$	set of left neighbors of plaintext chunk M
$\mathcal{R}_C$	set of right neighbors of ciphertext chunk C
$\mathcal{R}_M$	set of right neighbors of plaintext chunk M
$\mathcal{G}$	set of currently inferred ciphertext-plaintext chunk pairs
u	number of ciphertext-plaintext chunk pairs returned from frequency analysis during the initialization of $\mathcal{G}$
v	number of ciphertext-plaintext chunk pairs returned from frequency analysis in each iteration of locality-based attack
w	maximum size of $\mathcal{G}$
$\mathbf{L_C}$	associative array that maps each ciphertext chunk in $\mathbf{C}$ to its left neighbor and co-occurrence frequency
$\mathbf{L_M}$	associative array that maps each plaintext chunk in $\mathbf{M}$ to its left neighbor and co-occurrence frequency
$\mathbf{R_C}$	associative array that maps each ciphertext chunk in $\mathbf{C}$ to its right neighbor and co-occurrence frequency
$\mathbf{R_M}$	associative array that maps each plaintext chunk in $\mathbf{M}$ to its right neighbor and co-occurrence frequency
Defined in Section 6	
K_S	segment-based key of segment S
h	minimum fingerprint of chunks in a segment

4.1 Basic Attack

We first demonstrate how we can apply frequency analysis to infer the original plaintext chunks of the latest backup in encrypted deduplication. We call this attack the *basic attack*. Note that the basic attack is ineffective in inference (see below and the evaluation in Section 5), yet we use it to guide our design of the locality-based attack in Section 4.2.

Overview: In the basic attack, we identify each chunk by its fingerprint, and count the frequency of each chunk by the number of fingerprints that appear in a backup. Thus, a chunk (or a fingerprint) has a high frequency if there exist many identical chunks with the same content. We sort the chunks of both $\mathbf{C}$ and $\mathbf{M}$ by their frequencies, and infer that the i-th frequent plaintext chunk in $\mathbf{M}$ is the original plaintext chunk of the i-th frequent ciphertext chunk in $\mathbf{C}$. Our rationale is that the frequency of a plaintext chunk is correlated to that of the corresponding ciphertext chunk due to deterministic encryption.

Algorithm details: Algorithm 1 shows the pseudo-code of the basic attack. It takes $\mathbf{C}$ and $\mathbf{M}$ as input, and returns the result set $\mathcal{T}$ of all inferred ciphertext-plaintext chunk pairs. It first calls the function COUNT to obtain the frequencies of all ciphertext and plaintext chunks, identified by fingerprints, in associative arrays $\mathbf{F_C}$ and $\mathbf{F_M}$, respectively (Lines 2-3). It then calls the function FREQ-ANALYSIS to infer the set $\mathcal{T}$ of ciphertext-plaintext chunk pairs (Line 4), and returns $\mathcal{T}$ (Line 5).

The function COUNT constructs an associative array $\mathbf{F_X}$ (where $\mathbf{X}$ can be either $\mathbf{C}$ and $\mathbf{M}$) that holds

---
Algorithm 1 Basic Attack

1: **procedure** BASIC ATTACK($\mathbf{C}, \mathbf{M}$)
2:     $\mathbf{F_C} \leftarrow$ COUNT($\mathbf{C}$)
3:     $\mathbf{F_M} \leftarrow$ COUNT($\mathbf{M}$)
4:     $\mathcal{T} \leftarrow$ FREQ-ANALYSIS($\mathbf{F_C}, \mathbf{F_M}$)
5:     **return** $\mathcal{T}$
6: **end procedure**

7: **function** COUNT($\mathbf{X}$)
8:     Initialize $\mathbf{F_X}$
9:     **for** each X in $\mathbf{X}$ **do**
10:         **if** X does not exist in $\mathbf{F_X}$ **then**
11:             Initialize $\mathbf{F_X}[X] \leftarrow 0$
12:         **end if**
13:         $\mathbf{F_X}[X] \leftarrow \mathbf{F_X}[X] + 1$
14:     **end for**
15:     **return** $\mathbf{F_X}$
16: **end function**

17: **function** FREQ-ANALYSIS($\mathbf{F_C}, \mathbf{F_M}$)
18:     Sort $\mathbf{F_C}$ by frequency
19:     Sort $\mathbf{F_M}$ by frequency
20:     $min \leftarrow \min\{|\mathbf{F_C}|, |\mathbf{F_M}|\}$
21:     **for** $i = 1$ to min **do**
22:         $C \leftarrow i$-th frequent ciphertext chunk
23:         $M \leftarrow i$-th frequent plaintext chunk
24:         Add (C, M) to $\mathcal{T}'$
25:     **end for**
26:     **return** $\mathcal{T}'$
27: **end function**
---

the frequencies of all chunks. If a chunk X does not exist in $\mathbf{F_X}$ (i.e., its fingerprint is not found), then the function adds X to $\mathbf{F_X}$ and initializes $\mathbf{F_X}[X]$ as zero (Lines 10-12). The function then increments $\mathbf{F_X}[X]$ by one (Line 13).

The function FREQ-ANALYSIS performs frequency analysis based on $\mathbf{F_C}$ and $\mathbf{F_M}$. It first sorts each of $\mathbf{F_C}$ and $\mathbf{F_M}$ by frequency (Lines 18-19). Since $\mathbf{F_C}$ and $\mathbf{F_M}$ may not have the same number of elements, it finds the minimum number of elements in $\mathbf{F_C}$ and $\mathbf{F_M}$ (Line 20). Finally, it returns the ciphertext-plaintext chunk pairs, in which both the ciphertext and plaintext chunks of each pair have the same rank (Lines 21-26).

Discussion: The basic attack demonstrates how frequency analysis can be applied to encrypted deduplication. However, it only achieves a small inference rate, as shown in our trace-driven evaluation (see Section 5). One reason is that the basic attack is sensitive to data updates that occur across different versions of backups over time. An update to a chunk can change the frequency ranks of multiple chunks, including the chunk itself and other chunks with similar frequencies. Specifically, an update to a ciphertext chunk C in the latest backup can lower the frequency rank of C, which now has fewer copies of identical ciphertext chunks with the same content, while promoting the ranks of other ciphertext chunks whose ranks are just below C. Another reason is that there exist many ties, in which chunks have the same frequency. How to break a tie during sorting also affects the frequency rank and hence the inference results of the tied chunks. In the following, we extend the basic attack to improve its inference rate.

4.2 Locality-based Attack

The *locality-based attack* exploits *chunk locality* to make frequency analysis more effective.

Overview: We first define the notation that captures the notion of chunk locality. Consider two ordered pairs $\langle C_i, C_{i+1} \rangle$ and $\langle M_i, M_{i+1} \rangle$ of neighboring ciphertext and plaintext chunks in **C** and **M**, respectively. We say that C_i is the *left* neighbor of C_{i+1}, while C_{i+1} is the *right* neighbor of C_i; similar definitions apply to M_i and M_{i+1}. Note that a ciphertext chunk in **C** or a plaintext chunk in **M** may repeat many times (i.e., there are many duplicate copies), so if we identify each chunk by its fingerprint, it can be associated with more than one left or right neighbor. Let $\mathcal{L}_C$ and $\mathcal{R}_C$ be the sets of left neighbors and right neighbors of a ciphertext chunk C, respectively, and $\mathcal{L}_M$ and $\mathcal{R}_M$ be the left and right neighbors of a plaintext chunk M, respectively.

Our insight is that if a plaintext chunk M of a prior backup has been identified as the original plaintext chunk of a ciphertext chunk C of the latest backup, then the left and right neighbors of M are also likely to be original plaintext chunks of the left and right neighbors of C, mainly because chunk locality implies that the ordering of chunks is likely to be preserved across backups. In other words, for any inferred ciphertext-plaintext chunk pair (C, M), we further infer more ciphertext-plaintext chunk pairs through the left and right neighboring chunks of C and M, and repeat the same inference on those newly inferred chunk pairs. Thus, we can significantly increase the attack severity.

The locality-based attack operates on an *inferred set* $\mathcal{G}$, which stores the currently inferred set of ciphertext-plaintext chunks pairs. How to initialize $\mathcal{G}$ depends on the attack modes (see Section 3). In ciphertext-only mode, in which an adversary only knows **C** and **M**, we apply frequency analysis to find the most frequent ciphertext-plaintext chunk pairs and add them to $\mathcal{G}$. Here, we configure a parameter u (e.g., $u = 1$ by default in our implementation) to indicate the number of most frequent chunk pairs to be returned. Our rationale is that the top-frequent chunks have significantly higher frequencies (see Figure 1) than the other chunks, and their frequency ranks are stable across different backups. This ensures the correctness of the ciphertext-plaintext chunk pairs in $\mathcal{G}$ with a high probability throughout the attack. On the other hand, in known-plaintext mode, in which the adversary knows some leaked ciphertext-plaintext chunk pairs about **C** for the latest backup, we initialize $\mathcal{G}$ with the set of leaked chunk pairs that are also in **M**.

The locality-based attack proceeds as follows. In each iteration, it picks one ciphertext-plaintext chunk pair (C, M) from $\mathcal{G}$. It collects the corresponding sets of neighboring chunks $\mathcal{L}_C$, $\mathcal{L}_M$, $\mathcal{R}_C$, and $\mathcal{R}_M$. We apply frequency analysis to find the most frequent ciphertext-plaintext chunk pairs from each of $\mathcal{L}_C$ and $\mathcal{L}_M$, and similarly from $\mathcal{R}_C$ and $\mathcal{R}_M$. In other words, we find the left and right neighboring chunks of C and M that have the most *co-occurrences* with C and M themselves, respectively. We configure a parameter v (e.g., $v = 15$ by default in our implementation) to indicate the number of most frequent chunk pairs returned from the frequent analysis algorithm being called in each iteration. A larger v increases the number of inferred ciphertext-plaintext chunk pairs, but it also potentially compromises the inference accuracy. The attack adds all inferred chunk pairs into $\mathcal{G}$, and iterates until all inferred chunk pairs in $\mathcal{G}$ have been processed.

Note that $\mathcal{G}$ may grow very large as the backup size increases. A very large $\mathcal{G}$ can exhaust memory space. We configure a parameter w (e.g., $w = 200{,}000$ by default in our implementation) to bound the maximum size of $\mathcal{G}$.

In our evaluation (see Section 5), we carefully examine the impact of the configurable parameters u, v, and w.

Algorithm details: Algorithm 2 shows the pseudo-code of the locality-based attack. It takes **C**, **M**, u, v, and w as input, and returns the result set $\mathcal{T}$ of all inferred ciphertext-plaintext chunk pairs. It first calls the function COUNT to obtain the following associative arrays: $\mathbf{F_C}$, which stores the frequencies of all ciphertext chunks, as well as $\mathbf{L_C}$ and $\mathbf{R_C}$, which store the co-occurrence frequencies of the left and right neighbors of all ciphertext chunks, respectively (Line 2); similarly, it obtains the associative arrays $\mathbf{F_M}$, $\mathbf{L_M}$, and $\mathbf{R_M}$ for the plaintext chunks (Line 3). It then initializes the inferred set $\mathcal{G}$, either by obtaining u most

Algorithm 2 Locality-based Attack

1: **procedure** LOCALITY ATTACK($\mathbf{C}, \mathbf{M}, u, v, w$)
2:     $(\mathbf{F_C}, \mathbf{L_C}, \mathbf{R_C}) \leftarrow$ COUNT($\mathbf{C}$)
3:     $(\mathbf{F_M}, \mathbf{L_M}, \mathbf{R_M}) \leftarrow$ COUNT($\mathbf{M}$)
4:     **if** ciphertext-only mode **then**
5:         $\mathcal{G} \leftarrow$ FREQ-ANALYSIS($\mathbf{F_C}, \mathbf{F_M}, u$)
6:     **else if** known-plaintext mode **then**
7:         $\mathcal{G} \leftarrow$ set of leaked ciphertext-plaintext chunk pairs that appear in both $\mathbf{C}$ and $\mathbf{M}$
8:     **end if**
9:     $\mathcal{T} \leftarrow \mathcal{G}$
10:     **while** $\mathcal{G}$ is non-empty **do**
11:         Remove (C, M) from $\mathcal{G}$
12:         $\mathcal{T}_l \leftarrow$ FREQ-ANALYSIS($\mathbf{L_C}[C], \mathbf{L_M}[M], v$)
13:         $\mathcal{T}_r \leftarrow$ FREQ-ANALYSIS($\mathbf{R_C}[C], \mathbf{R_M}[M], v$)
14:         **for** each (C, M) in $\mathcal{T}_l \cup \mathcal{T}_r$ **do**
15:             **if** $(C, *)$ is not in $\mathcal{T}$ **then**
16:                 Add (C, M) to $\mathcal{T}$
17:                 **if** $|\mathcal{G}| \leq w$ (i.e., $\mathcal{G}$ is not full) **then**
18:                     Add (C, M) to $\mathcal{G}$
19:                 **end if**
20:             **end if**
21:         **end for**
22:     **end while**
23:     **return** $\mathcal{T}$
24: **end procedure**

25: **function** COUNT($\mathbf{X}$)
26:     Initialize $\mathbf{F_X}, \mathbf{L_X}$, and $\mathbf{R_X}$
27:     **for** each X in $\mathbf{X}$ **do**
28:         **if** X does not exist in $\mathbf{F_X}$ **then**

29:         Initialize $\mathbf{F_X}[X] \leftarrow 0$
30:         **end if**
31:     $\mathbf{F_X}[X] \leftarrow \mathbf{F_X}[X] + 1$
32:     **if** X has a left neighbor X_l **then**
33:         **if** X_l does not exist in $\mathbf{L_X}[X]$ **then**
34:             Initialize $\mathbf{L_X}[X][X_l] \leftarrow 0$
35:         **end if**
36:         $\mathbf{L_X}[X][X_l] \leftarrow \mathbf{L_X}[X][X_l] + 1$
37:     **end if**
38:     **if** X has a right neighbor X_r **then**
39:         **if** X_r does not exist in $\mathbf{R_X}[X]$ **then**
40:             Initialize $\mathbf{R_X}[X][X_r] \leftarrow 0$
41:         **end if**
42:         $\mathbf{R_X}[X][X_r] \leftarrow \mathbf{R_X}[X][X_r] + 1$
43:     **end if**
44:     **end for**
45:     **return** $(\mathbf{F_X}, \mathbf{L_X}, \mathbf{R_X})$
46: **end function**

47: **function** FREQ-ANALYSIS($\mathbf{Y_C}, \mathbf{Y_M}, x$)
48:     Sort $\mathbf{Y_C}$ by frequency
49:     Sort $\mathbf{Y_M}$ by frequency
50:     **for** $i = 1$ to $\min\{x, |\mathbf{Y_C}|, |\mathbf{Y_M}|\}$ **do**
51:         $C \leftarrow i$-th frequent ciphertext chunk
52:         $M \leftarrow i$-th frequent plaintext chunk
53:         Add (C, M) to $\mathcal{T}'$
54:     **end for**
55:     **return** $\mathcal{T}'$
56: **end function**

---

frequent ciphertext-plaintext chunk pairs from frequency analysis in ciphertext-only mode, or by adding the set of leaked ciphertext-plaintext chunk pairs that appear in both the latest and prior backups (i.e., $\mathbf{C}$ and $\mathbf{M}$, respectively) in known-plaintext mode (Lines 4-8). It also initializes $\mathcal{T}$ with $\mathcal{G}$ (Line 9).

In the main loop (Lines 10-22), the algorithm removes a pair (C, M) from $\mathcal{G}$ (Line 11) and uses it to infer additional ciphertext-plaintext chunk pairs from the neighboring chunks of C and M. It first examines all left neighbors by running the function FREQ-ANALYSIS on $\mathbf{L_C}[C]$ and $\mathbf{L_M}[M]$, and stores v most frequent ciphertext-plaintext chunk pairs in $\mathcal{T}_l$ (Line 12). Similarly, it examines all right neighbors and stores the results in $\mathcal{T}_r$ (Line 13). For each (C, M) in $\mathcal{T}_l \cup \mathcal{T}_r$, if $(C, *)$ is not in $\mathcal{T}$ (i.e., the original plaintext chunk of C has not been inferred yet), we add (C, M) to $\mathcal{T}$ and also to $\mathcal{G}$ if $\mathcal{G}$ is not full (Lines 14-21). The main loop iterates until $\mathcal{G}$ becomes empty. Finally, $\mathcal{T}$ is returned.

Both the functions COUNT and FREQ-ANALYSIS are similar to those in the basic attack (see Algorithm 1), with the following extensions. For COUNT, in addition to constructing the associative array $\mathbf{F_X}$ (where $\mathbf{X}$ can be either $\mathbf{C}$ and $\mathbf{M}$) that holds the frequencies of all chunks, it also constructs the associative arrays $\mathbf{L_X}$ and $\mathbf{R_X}$ that hold the co-occurrence frequencies of the left and right neighbors of each chunk X, respectively. For FREQ-ANALYSIS, it now performs frequency analysis on the associative arrays $\mathbf{Y_C}$ and $\mathbf{Y_M}$, in which $\mathbf{Y_C}$ (resp. $\mathbf{Y_M}$) refers to either $\mathbf{F_C}$ (resp. $\mathbf{F_M}$) that holds the frequency counts of all chunks, or $\mathbf{L_C}[C]$ and $\mathbf{R_C}[C]$ (resp. $\mathbf{L_M}[M]$ and $\mathbf{R_M}[M]$) that hold the frequency counts of all ordered pairs of chunks associated with ciphertext chunk C (resp. plaintext chunk M). Also, FREQ-ANALYSIS only returns
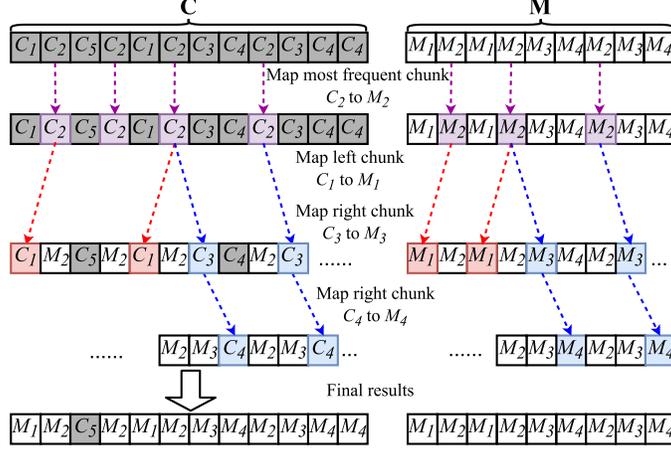
Figure 3: Example of the locality-based attack.

x (where x can be either u or v) most frequent ciphertext-plaintext chunk pairs.

Example: Figure 3 shows an example of how the locality-based attack works. Here, we consider ciphertext-only mode. Suppose that we have obtained the auxiliary information $\mathbf{M} = \langle M_1, M_2, M_1, M_2, M_3, M_4, M_2, M_3, M_4 \rangle$ of some prior backup, and use it to infer the original plaintext chunks of $\mathbf{C} = \langle C_1, C_2, C_5, C_2, C_1, C_2, C_3, C_4, C_2, C_3, C_4, C_4 \rangle$ of the latest backup. We set $u = v = 1$, and $w \to \infty$ (i.e., the inferred set $\mathcal{G}$ is unbounded). We assume that the ground truth is that the original plaintext chunk of the ciphertext chunk C_i is M_i for $i = 1, 2, 3, 4$, while that of C_5 is some new plaintext chunk not in $\mathbf{M}$ (note that in reality, an adversary does not know the ground truth).

We first apply frequency analysis and find that (C_2, M_2) is the most frequent ciphertext-plaintext chunk pair, so we initialize $\mathcal{G} = \{(C_2, M_2)\}$ and add it into $\mathcal{T}$. We then remove and operate on (C_2, M_2) from $\mathcal{G}$, and find that $\mathcal{L}_{C_2} = \{C_1, C_4, C_5\}$, $\mathcal{L}_{M_2} = \{M_1, M_4\}$, $\mathcal{R}_{C_2} = \{C_1, C_3, C_5\}$, and $\mathcal{R}_{M_2} = \{M_1, M_3\}$. From $\mathcal{L}_{C_2}$ and $\mathcal{L}_{M_2}$, we find that (C_1, M_1) is the most frequent ciphertext-plaintext chunk pair, while from $\mathcal{R}_{C_2}$ and $\mathcal{R}_{M_2}$, we find (C_3, M_3). Thus, we add both (C_1, M_1) and (C_3, M_3) into $\mathcal{G}$ and $\mathcal{T}$. We repeat the processing on (C_1, M_1) and (C_3, M_3), and we can infer another pair (C_4, M_4) from the right neighbors of (C_3, M_3).

To summarize, the locality-based attack can successfully infer the original plaintext chunks of all four ciphertext chunks C_1, C_2, C_3, and C_4. It cannot infer the original plaintext chunk of C_5, as it does not appear in $\mathbf{M}$.

4.3 Advanced Locality-based Attack

Based on the framework of the locality-based attack, we propose an advanced locality-based attack that specifically targets variable-size chunks generated from content-defined chunking (see Section 2.1). Specifically, if the generated chunks have varying sizes, the adversary can observe the size of each ciphertext chunk before deduplication and leverage the size information to increase the severity of the locality-based attack.

Overview: The advanced locality-based attack builds on the observation that if a ciphertext chunk C corresponds to a plaintext chunk M, then the actual size of C approximates that of M. Suppose that the symmetric encryption algorithm used by the encrypted deduplication system is based on block ciphers (e.g., AES), then both C and M should have the same number of blocks (i.e., the basic units of block ciphers). We exploit this additional information in frequency analysis. Specifically, we first classify the sets of ciphertext chunks (i.e., $\mathbf{C}$, $\mathcal{L}_C$, $\mathcal{R}_C$) and plaintext chunks (i.e., $\mathbf{M}$, $\mathcal{L}_M$, and $\mathcal{R}_M$) by their sizes, measured

---

Algorithm 3 Frequency Analysis in the Advanced Locality-based Attack

---

1: **function** FREQ-ANALYSIS($\mathbf{Y_C}, \mathbf{Y_M}, x$)
2:     $\mathbf{B_C} \leftarrow$ CLASSIFY($\mathbf{Y_C}$)
3:     $\mathbf{B_M} \leftarrow$ CLASSIFY($\mathbf{Y_M}$)
4:     **for** each available size s **do**
5:         Sort $\mathbf{B_C}[s]$ by frequency
6:         Sort $\mathbf{B_M}[s]$ by frequency
7:         **for** $i = 1$ to x **do**
8:             $C \leftarrow i$-th frequent ciphertext chunk in $\mathbf{B_C}[s]$
9:             $M \leftarrow i$-th frequent plaintext chunk in $\mathbf{B_M}[s]$
10:            Add (C, M) to $\mathcal{T}'$
11:         **end for**
12:     **end for**
13:     **return** $\mathcal{T}'$
14: **end function**

15: **function** CLASSIFY($\mathbf{Y_X}$)
16:     Initialize $\mathbf{B_X}$
17:     **for** each X in Y_X **do**
18:         $s \leftarrow \lceil \frac{\text{size}(X)}{16} \rceil$
19:         **if** s does not exist in $\mathbf{B_X}$ **then**
20:            Initialize $\mathbf{B_X}[s] \leftarrow 0$
21:         **end if**
22:         $\mathbf{B_X}[s][X] \leftarrow \mathbf{Y_X}[X]$
23:     **end for**
24:     **return** $\mathbf{B_X}$
25: **end function**

---

in terms of the number of blocks. For each available chunk size, we relate top-frequent ciphertext chunks with the top-frequent plaintext chunks that have the same size. This improves the accuracy of each inferred ciphertext-plaintext pair, and hence the inferred neighbors in the iterated inference of the locality-based attack.

Algorithm details: The advanced locality-based attack extends the original locality-based attack in Algorithm 2 and modifies the function FREQ-ANALYSIS (called in Line 5 and Lines 12-13 in Algorithm 2) to augment frequency analysis with the knowledge of chunk sizes.

Algorithm 3 shows the pseudo-code of frequency analysis in the advanced locality-based attack. As in Algorithm 2, the function FREQ-ANALYSIS takes the associative arrays $\mathbf{Y_C}$ and $\mathbf{Y_M}$, as well as the parameter x, as input. It calls the function CLASSIFY to classify the ciphertext and plaintext chunks in $\mathbf{Y_C}$ and $\mathbf{Y_M}$ into $\mathbf{B_C}$ and $\mathbf{B_M}$, respectively (Lines 2-3), where $\mathbf{B_C}$ (resp. $\mathbf{B_M}$) maps the ciphertext (resp. plaintext) chunks that have the same sizes to corresponding frequencies. It infers x top-frequent ciphertext-plaintext pairs for each available s (Lines 4-12), and finally returns the inference results (Line 13).

The function CLASSIFY groups the chunks in $\mathbf{Y_X}$ (where $\mathbf{X}$ can be either $\mathbf{C}$ or $\mathbf{M}$) by their sizes. In this work, we assume that AES encryption is used and the block size is 16 bytes. Thus, CLASSIFY derives the number of blocks s of each ciphertext or plaintext chunk (denoted by X) (Line 18), and stores the frequency of X in $\mathbf{B_X}[s][X]$ (Line 22).

5 Attack Evaluation

We present trace-driven evaluation results on the severity of frequency analysis against encrypted dedupli-cation.

5.1 Datasets

We consider three datasets in our evaluation.

FSL: This dataset is collected by the File systems and Storage Lab (FSL) at Stony Brook University [1, 59, 60] and describes real-world storage patterns. We focus on the *Fslhomes* dataset, which contains the daily snapshots of users' home directories on a shared file system. Each snapshot is represented by a collection of 48-bit chunk fingerprints produced by variable-size chunking of different average sizes. We pick the snapshots from January 22 to May 21 in 2013, and fix the average size as 8 KB for our evaluation. We select six users (User4, User7, User12, User13, User15, and User28) that have the complete daily snapshots over the whole duration. We aggregate each user's snapshots on a monthly basis (on January 22, February 22, March 22, April 21, and May 21), and hence form five monthly full backups for all users. Our post-processed dataset covers a total of 2.7 TB of logical data before deduplication, and the overall deduplication ratio (i.e., the ratio of the logical data size to the physical data size after deduplication) is $7.6\times$.

Synthetic: This dataset contains a sequence of synthetic backup snapshots that are generated based on Lillibridge *et al.*'s approach [44]. Specifically, we create an initial snapshot from a Ubuntu 14.04 virtual disk image (originally with 1.1 GB of data) with a total of 4.3 GB space. We create a sequence of snapshots starting from the initial snapshot, such that each snapshot is created from the previous one by randomly picking 2% of files and modifying 2.5% of their content, and also adding 10 MB of new data. Finally, we generate a sequence of ten snapshots, each of which is treated as a backup. Based on our choices of parameters, the resulting storage saving of deduplication is around 90% (or equivalently, the deduplication ratio is around $10\times$, which is typical in real-life backup workloads [62]). Note that the initial snapshot is publicly available. Later in our evaluation, we study the effectiveness of the attacks by using initial snapshot as the public auxiliary information.

VM: This dataset is collected by ourselves in a real-world scenario. It comprises 156 virtual machine (VM) image snapshots for the students enrolled in a university programming course in Spring 2014. Each snapshot is represented by the SHA-1 fingerprints of 4 KB fixed-size chunks. We treat the VM image snapshot as a weekly backup of a user, and extract 13 weeks of backups of all users. We remove all zero-filled chunks that dominate in VM images [34], and obtain a reduced dataset covering 9.11TB of data. The overall deduplication ratio of the VM dataset is $47.6\times$. Our prior studies [43, 53] have also used the variants of the dataset for evaluation. Here, we include this dataset for cross-validation of other datasets in our attack evaluation.

5.2 Methodology

We implement all three inference attacks by processing and comparing the chunk fingerprints in our datasets. We benchmark our current implementation on a Ubuntu 16.04 Linux machine with an AMD Athlon II X4 640 quad-core 3.0 GHz CPU and 16 GB RAM. In general, both the memory usage and the processing time of the locality-based attack increase with the total number of unique chunks in a backup (the actual performance overhead of the locality-based attack depends on the number of inferred chunk pairs being processed in each iteration). Specifically, the locality-based attack takes around 1.8 GB memory and 15 hours to process an FSL backup that includes around 30 million unique chunks, and takes around 600 MB memory and 10 hours to process a VM backup that includes around 6 million unique chunks. In the following, we highlight the implementation details of some data structures used by the attacks.
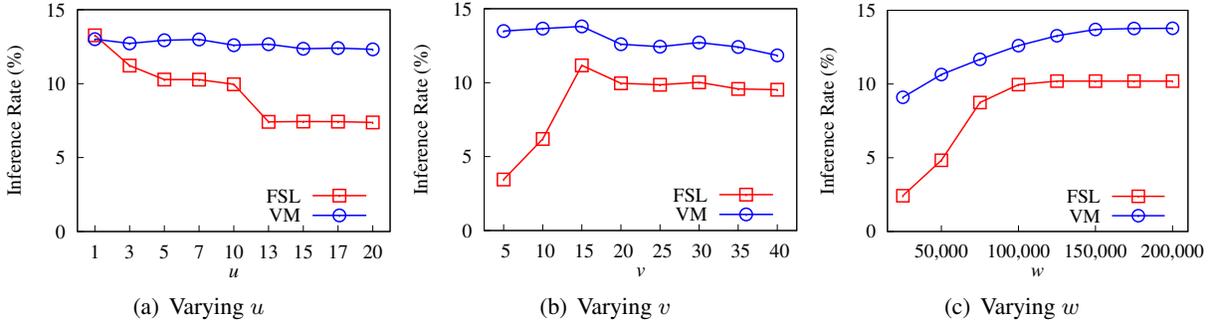
Figure 4: Attack evaluation: Impact of parameters on locality-based attack.

Associative arrays: Recall that there are three types of associative arrays: (i) $\mathbf{F_C}$ and $\mathbf{F_M}$, (ii) $\mathbf{L_C}$ and $\mathbf{L_M}$, and (iii) $\mathbf{R_C}$ and $\mathbf{R_M}$ (the latter two are only used by the locality-based attack). We implement them as key-value stores using LevelDB [26]. Each key-value store is keyed by the fingerprint of the ciphertext/plaintext chunk. For $\mathbf{F_C}$ and $\mathbf{F_M}$, each entry stores a frequency count; for $\mathbf{L_C}$, $\mathbf{L_M}$, $\mathbf{R_C}$, and $\mathbf{R_M}$, each entry stores a sequential list of the fingerprints of all the left/right neighbors of the keyed chunk and the co-occurrence frequency counts. For the latter, keeping neighboring chunks sequentially simplifies our implementation, but also increases the search time of a particular neighbor (which dominates the overall running time); we pose the optimization as future work.

Inferred set: We implement the inferred set $\mathcal{G}$ in the locality-based attack as a first-in-first-out queue, whose maximum size is bounded by w (see Section 4.2). Each time we remove the first ciphertext-plaintext chunk pair from the queue for inferring more chunk pairs from the neighbors.

5.3 Results

We now present the evaluation results and show the inference rate (defined in Section 4) of each attack under different settings.

5.3.1 Impact of Parameters

We first evaluate the impact of parameters on the locality-based attack, in order to justify our choices of parameters. Recall that the locality-based attack is configured with three parameters: u, v, and w, where u and v are the numbers of ciphertext-plaintext pairs returned by frequency analysis during the initialization of $\mathcal{G}$ and each iteration of the locality-based attack, respectively, and w is the maximum size of $\mathcal{G}$. Here, we focus on the FSL and VM datasets, and evaluate the attack in ciphertext-only mode. For the FSL dataset, we use the backup on March 22 as the auxiliary information in order to infer the plaintext chunks of the latest backup on May 21; for the VM dataset, we use the 12th weekly backup to infer the plaintext chunks of the latest 13th weekly backup.

Figure 4(a) first shows the impact of u, in which we fix $v = 20$ and $w = 100,000$. The inference rate gradually decreases with u. For example, when u increases from 1 to 20, the inference rate decreases from 13.3% to 7.4% and from 13.0% to 12.3% for the FSL and VM datasets, respectively. A larger u implies that incorrect ciphertext-plaintext chunk pairs are more likely to be included into the inferred set during initialization, thereby compromising the inference accuracy. In addition, the decrease of the inference rate in the VM dataset is slower than that in the FSL dataset. The reason is that we use a more recent VM backup as the auxiliary information and its frequency ranking is similar to that of the latest backup.

Figure 4(b) next shows the impact of v, in which we fix $u = 10$ and $w = 100,000$. Initially, the inference rate increases with v, as the underlying frequency analysis infers more ciphertext-plaintext chunk pairs in

15
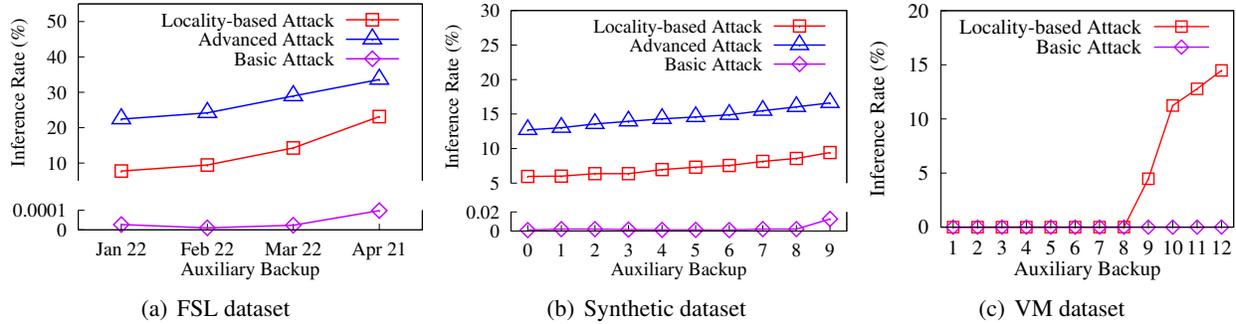
Figure 5: Attack evaluation: Inference rate in ciphertext-only mode (varying auxiliary backups, fixed target backup). Note that the locality-based attack and the advanced locality-based attack are equivalent for the VM dataset.

each iteration. It hits the maximum value at about 11.2% (for the FSL dataset) and 13.8% (for the VM dataset) when $v = 15$. When v increases to 40, the inference rate drops slightly to about 9.5% and 11.8% for the FSL and VM datasets, respectively. The reason is that some incorrectly inferred ciphertext-plaintext chunk pairs are also included into $\mathcal{G}$, which compromises the inference rate.

Figure 4(c) finally shows the impact of w, in which we fix $u = 10$ and $v = 20$. A larger w increases the inference rate, since $\mathcal{G}$ can hold more ciphertext-plaintext chunk pairs across iterations. We observe that when w increases beyond 200,000, the inference rate becomes steady at about 10.2% and 13.8% for the FSL and VM datasets, respectively.

5.3.2 Inference Rate in Ciphertext-only Mode

We now evaluate the attacks in ciphertext-only mode. We select $u = 1$, $v = 15$, and $w = 200,000$ as default parameters to achieve the highest possible inference rate.

Varying auxiliary backups: First, we choose each of the prior backups as the auxiliary information (we call them *auxiliary backups*), and infer the original plaintext chunks in the latest backup (i.e., the backup on May 21 in the FSL dataset, the 10th backup in the synthetic dataset, and the 13th backup in the VM dataset).

Figure 5 shows the inference rates of the attacks versus different auxiliary backups for different datasets (note that the zeroth auxiliary backup in the synthetic dataset is the publicly available image snapshot). As expected, the inference rates of all attacks increase as we use more recent auxiliary backups, which generally have higher content redundancy with the target latest backup. The basic attack is ineffective in all cases, as the inference rate is no more than 0.0001% for the FSL dataset, 0.02% for the synthetic dataset, and 0.005% for the VM dataset. In contrast, both the locality-based attack and the advanced locality-based attack achieve significantly high inference rates. For example, if we use the most recent prior backup (e.g, the FSL backup on April 21 and the 9th synthetic backup) as the auxiliary information, the inference rates of the locality-based attack and the advanced locality-based attack reach as high as 23.2% and 33.6% for the FSL dataset, as well as 9.4% and 16.6% for the synthetic dataset, respectively.

For the VM dataset, the locality-based attack and the advanced locality-based attack are equivalent, since all chunks have the same size. We observe that when we use the first eight VM backups as the auxiliary information, the inference rate of the locality-based attack is low (less than 0.005%). The possible reason is that users have heavy activities during these weeks, such that these prior backups have low content redundancy with the target latest backup. After the 8th backup, the inference rate of the locality-based attack increases and finally achieves 14.5%.

Varying target backups: We next fix the backup on January 22 in the FSL dataset, the initial Ubuntu image

16
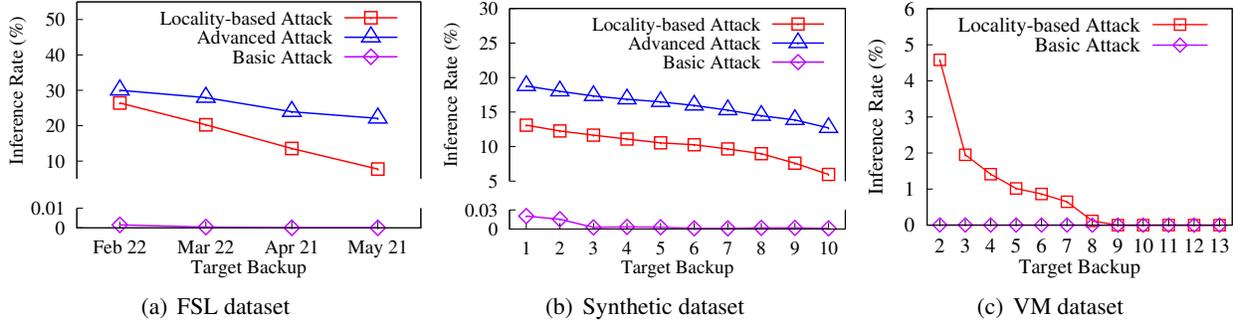
Figure 6: Attack evaluation: Inference rate in ciphertext-only mode (fixed auxiliary backup, varying target backups).

snapshot in the synthetic dataset, and the first backup in the VM dataset as the auxiliary information, and infer the original plaintext chunks in each of the following backups (we call them *target backups*).

Figure 6 shows the inference rates of the attacks versus different target backups for different datasets. Both the locality-based attack and the advanced locality-based attack are again more severe than the basic attack, whose inference rate is less than 0.03%. For example, the locality-based attack and the advanced locality-based attack can reach the inference rates of 26.4% and 30.0% for the backup on February 22 in the FSL dataset, as well as 13.1% and 18.8% for the first backup in the synthetic dataset, respectively. Since more chunks have been updated after a number of backups (e.g., four backups for the FSL dataset and ten backups for the synthetic dataset), the inference rates of the locality-based attack and the advanced locality-based attack drop to 7.7% and 22.1% for the FSL dataset, as well as 6.0% and 12.7% for the synthetic dataset, respectively.

For the VM dataset, the inference rate of the locality-based attack becomes low (e.g., around 0.1%) after using the 8th weekly backup as the target backup, since heavy updates appear during this period. Nevertheless, the locality-based attack still achieves a higher inference rate than the basic attack, whose inference rate is below 0.001%.

Attacks over a sliding window: Finally, we consider the launch of inference attacks based on a sliding window approach. Specifically, we choose the t-th backup as the auxiliary information, and infer the original plaintext chunks in the $(t + s)$-th backup, while we vary s and t in our evaluation. We mainly focus on the locality-based attack and the advanced locality-based attack, since the basic attack has low severity.

Figure 7 shows the inference rates for different s, where the x-axis represents different values of t. The advanced locality-based attack is more severe than the locality-based attack. For example, in the FSL dataset, the average inference rates of the locality-based attack are 24.3% and 17.3% for $s = 1$ and $s = 2$, while the corresponding inference rates of the advanced locality-based attack increase to 30.4% and 26.4%, respectively; in the synthetic dataset, the average inference rates of the locality-based attack are 12.0% and 11.3% for $s = 1$ and $s = 2$, while the corresponding inference rates of the advanced locality-based attack increase to 18.3% and 17.6%, respectively.

In addition, the inference rates of the VM dataset fluctuate significantly. For example, when we use the 3rd weekly backup as the auxiliary information, the inference rates hit the highest at 23.5%, 14.3%, and 14.4% for $s = 1$, 2, and 3, respectively. On the other hand, the inference rates drop down to less than 0.6% when we use the 5th to 8th weekly backups as the auxiliary information. Even with such non-preferable cases, the locality-based attack still achieves moderate severity in general, with an average inference rate of 12.5%, 7.3%, and 4.8% for $s = 1$, 2, and 3, respectively.
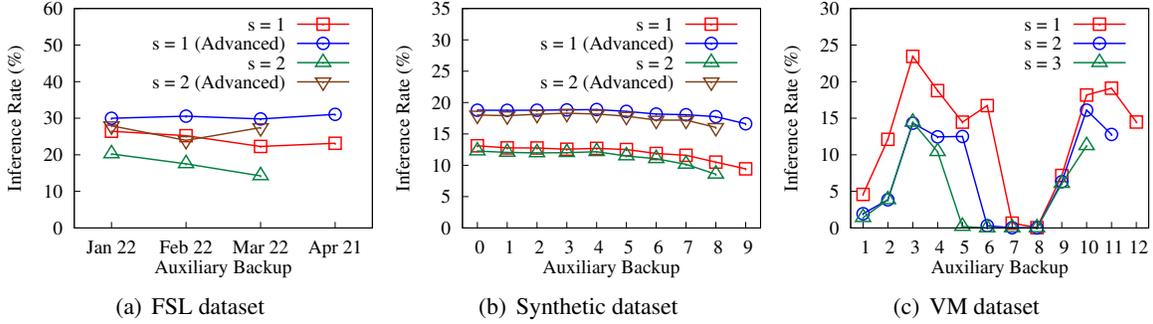
Figure 7: Attack evaluation: Inference rate in ciphertext-only mode (over a sliding window).
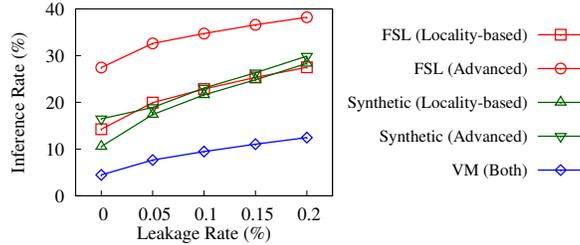
Figure 8: Attack evaluation: Inference rate in known-plaintext mode (varying leakage rates).

5.3.3 Inference Rate in Known-Plaintext Mode

We further evaluate the severity of the locality-based attack and the advanced locality-based attack in known-plaintext mode. To quantify the amount of leakage about the latest backup (see Section 3), we define the *leakage rate* as the ratio of the number of ciphertext-plaintext chunk pairs known by the adversary to the total number of ciphertext chunks in a target backup. We configure $u = 1$, $v = 15$, and $w = 500,000$. Note that we increase w to 500,000 (as opposed to $w = 200,000$ in Section 5.3.2), as the attack in known-plaintext mode can infer much more ciphertext-plaintext chunk pairs across iterations. Thus, we choose a larger w to include them into the inferred set.

Varying leakage rates: We fix both the auxiliary and target backups, and evaluate the inference rates of the attacks for different leakage rates. For the FSL dataset, we choose the backup on March 22 as the auxiliary information to infer the latest backup on May 21; for the synthetic dataset, we use the initial snapshot as the auxiliary information to infer the 5th backup snapshot; for the VM dataset, we use the 9th weekly backup as the auxiliary information to infer the 13th weekly backup.

Figure 8 shows the inference rates of the attacks (which also include the chunks that are already leaked in known-plaintext mode) for different leakage rates (varied from 0 to 0.2%) about the target backup being inferred. The slight increase in the leakage rate can lead to a significant increase in the inference rate. For example, when the leakage rate increases to 0.2%, the inference rates of the locality-based attack and the advanced locality-based attack reach 27.5% and 38.2% for the FSL dataset, and 28.3% and 29.9% for the synthetic dataset, respectively. Since the VM dataset includes fixed-size chunks, both attacks incur the same inference rate of 12.5% in this case.

Varying auxiliary backups: Based on the above setting, we consider different prior backups as the auxiliary information, while we fix the target backup as in the above experiment. We also fix the leakage rate as 0.05%.

Figure 9 shows the inference rates of the attacks for different auxiliary backups. We observe a similar tendency as in Figure 5. Specifically, in the FSL dataset, when using the FSL backup on April 21 as the auxiliary information, the inference rates of the locality-based attack and the advanced locality-based attack
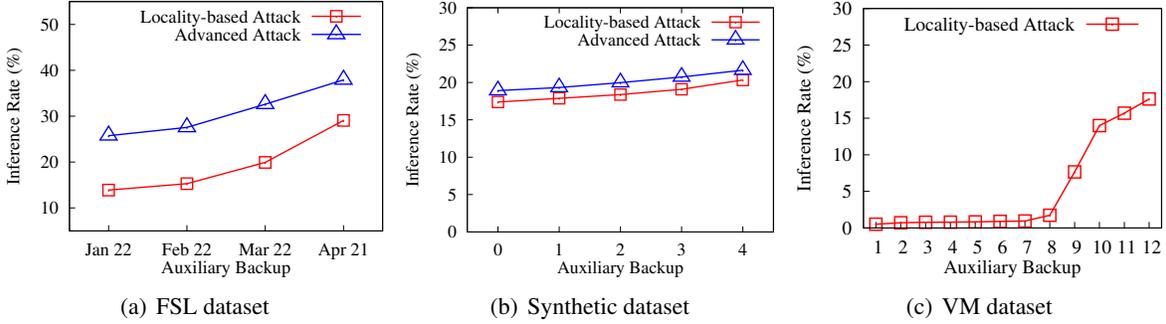
Figure 9: Attack evaluation: Inference rate in known-plaintext mode (varying auxiliary backups).

achieve 29.1% and 37.9%, respectively; in the synthetic dataset, when using the 4th synthetic backup as the auxiliary information, the corresponding inference rates achieve 20.3% and 21.6%, respectively; in the VM dataset, when using the 12th VM backup as the auxiliary information, the inference rate achieves 17.6% (same in both the locality-based attack and the advanced locality-based attack).

6 Defenses

The deterministic nature of encrypted deduplication discloses the frequency distribution of the underlying plaintext chunks, thereby making frequency analysis feasible. To defend against frequency analysis, we consider two defense approaches, namely *MinHash encryption* and *scrambling*.

6.1 MinHash Encryption

MinHash encryption builds on Broder's theorem [18], which states that if two sets share a large fraction of common elements (i.e., they are highly similar), then the probability that both sets share the same minimum hash element is also high. Since two backups from the same data source are expected to be highly similar and share a large number of identical chunks [62], MinHash encryption leverages this property to perform encrypted deduplication in a different way from the original MLE [12, 13]. We emphasize that previous deduplication approaches also leverage Broder's theorem to minimize the memory usage of the fingerprint index in plaintext deduplication [14,65] or key generation overhead in server-aided MLE [53]. Also, security analysis shows that MinHash encryption preserves data confidentiality as in server-aided MLE [53]. Thus, we do not claim the novelty of the design of MinHash encryption. Instead, our contribution is to study its effectiveness in defending against frequency analysis.

Algorithm 4 shows the pseudo-code of MinHash encryption, while we elaborate the implementation details in Section 7.1. MinHash encryption takes a sequence of plaintext chunks $\mathbf{M}$ as input, and returns a sequence of ciphertext chunks $\mathbf{C}$ as output. It partitions the plaintext chunks into *segments* (Line 3), each of which is a non-overlapped sub-sequence of adjacent plaintext chunks. For each segment S, MinHash encryption computes the minimum fingerprint h of all chunks in S, using the fingerprint value as the hash value of each chunk. It derives the segment-based key K_S based on h (Line 6), for example, by querying the key manager as in DupLESS [12] (see Section 2.2). It then encrypts each chunk in S using K_S and adds the resulting ciphertext chunk to $\mathbf{C}$ (Lines 7-10). Finally, it returns $\mathbf{C}$ (Line 12). Note that MinHash encryption only requests keys on a per-segment basis rather than on a per-chunk basis. As the number of segments is much less than that of chunks, the key generation overhead is greatly mitigated [53].

MinHash encryption is robust against the locality-based attack, by (slightly) breaking the deterministic nature of encrypted deduplication. Its rationale is that segments are highly similar as they share many

Algorithm 4 MinHash Encryption

```
 1: procedure MINHASH ENCRYPTION(M)
 2:     Initialize C
 3:     Partition M into segments
 4:     for each segment S do
 5:         h ← minimum fingerprint of all chunks in S
 6:         K_S ← segment-based key derived from h
 7:         for each chunk M ∈ S do
 8:             C ←ENCRYPT(K_S, M)
 9:             Add C into C
10:         end for
11:     end for
12:     return C
13: end procedure
```

identical plaintext chunks in backups [14, 65]. Thus, their minimum fingerprints, and hence the secret keys derived for segments, are likely to be the same as well due to Broder's theorem [18]. This implies that most identical plaintext chunks across segments are still encrypted by the same secret keys into identical ciphertext chunks, thereby preserving deduplication effectiveness. However, some identical plaintext chunks may still reside in different segments with different minimum fingerprints and hence different secret keys, so their resulting ciphertext chunks will be different and cannot be deduplicated, leading to a slight degradation of storage efficiency. Nevertheless, such "approximate" deduplication sufficiently alters the overall frequency ranking of ciphertext chunks by encrypting a small fraction of duplicate chunks using different keys, thereby making frequency analysis ineffective.

6.2 Scrambling

Scrambling augments MinHash encryption by disturbing the processing sequence of chunks, so as to prevent an adversary from correctly identifying the neighbors of each chunk in the locality-based attack. It is applied before the chunks are encrypted and stored, and its idea is to scramble the original plaintext chunk sequence $\mathbf{M}$ into a new sequence $\mathbf{M'}$. To be compatible with MinHash encryption, scrambling works on a per-segment basis by shuffling the ordering of chunks within each segment. Each plaintext chunk is still encrypted via MinHash encryption and stored as a ciphertext chunk, while the original file can still be reconstructed based on its file recipe and key recipe (see Section 2). Specifically, the file recipe contains a list of fingerprints that are stored in the original order of the plaintext chunks before scrambling. If a client wants to restore the original file, it first retrieves the file recipe and the key recipe (which are encrypted by the client's own secret key), followed by decrypting the ciphertext chunks (based on the key recipe) and restoring the original order of the plaintext chunks and hence the original file (based on the file recipe).

Note that scrambling does not change the storage efficiency of MinHash encryption, since it only changes the order of plaintext chunks. Also, we apply scrambling on a per-segment basis, while a deduplicated storage system typically organizes unique chunks in *containers* [44] that serve as the basic read/write units. In our prototype (see Section 7.4), we configure the container size larger than the segment size (e.g., we set the container size as 4 MB, while setting the maximum segment size as 2 MB). Thus, the scrambling approach has limited impact on the chunk layout across containers, so it does not add substantial overhead to the overall read/write performance. In Section 7.4, we will study how the scrambling approach affects the performance of a deduplicated storage system.

Algorithm 5 elaborates the pseudo-code of scrambling. It first partitions the original plaintext chunk sequence $\mathbf{M}$ into segments as in MinHash encryption (Line 2). Then for each chunk of a segment S, the

---

Algorithm 5 Scrambling

1: **procedure** SCRAMBLING(**M**)
2:     Initialize **M**′
3:     Partition **M** into segments
4:     **for** each segment S **do**
5:         Initialize S'
6:         **for** each chunk $M \in S$ **do**
7:             Generate a random number r
8:             **if** r is odd **then**
9:                 Add M as first chunk of S'
10:             **else**
11:                 Add M as last chunk of S'
12:             **end if**
13:         **end for**
14:         Add S' into **M**′
15:     **end for**
16:     **return M**′
17: **end procedure**

---

algorithm randomly adds the chunk to either the front of S' or the end of S', where S' is the scrambled version of S (Lines 6-13). Finally, it returns the scrambled sequence **M**′ that includes all the scrambled segments (Line 16).

7   Defense Evaluation

We conduct trace-driven evaluation on MinHash encryption and scrambling in three aspects: defense effectiveness, storage efficiency, and metadata access overhead.

7.1   Methodology

Since both FSL and VM datasets do not contain actual contents, we simulate our defense approaches by directly operating on chunk fingerprints. First, we identify segment boundaries based on chunk fingerprints, by following the variable-size segmentation scheme in [45]. Specifically, the segmentation scheme is configured by the minimum, average, and maximum segment sizes. It places a segment boundary at the end of a chunk fingerprint if (i) the size of each segment is at least the minimum segment size, and (ii) the chunk fingerprint modulo a pre-defined divisor (which determines the average segment size) is equal to some constant (e.g., -1), or the inclusion of the chunk makes the segment size larger than the maximum segment size. In our evaluation, we set the minimum, average, and maximum segment sizes as 512 KB, 1 MB, and 2 MB, respectively.

After scrambling the orders of chunks (a.k.a. fingerprints) in each segment, we mimic MinHash encryption as follows. We first calculate the minimum chunk fingerprint h of each segment. We then concatenate h with each chunk fingerprint in the segment and compute the SHA-256 hash of the concatenation. We also truncate the hash result to be consistent with the fingerprint sizes in the original FSL and VM datasets, respectively. The truncated hash result can be viewed as the fingerprint of the ciphertext chunk. We can easily check that identical plaintext chunks under the same h will lead to identical ciphertext chunks that can be deduplicated.

(a) FSL dataset   (b) Synthetic dataset   (c) VM dataset
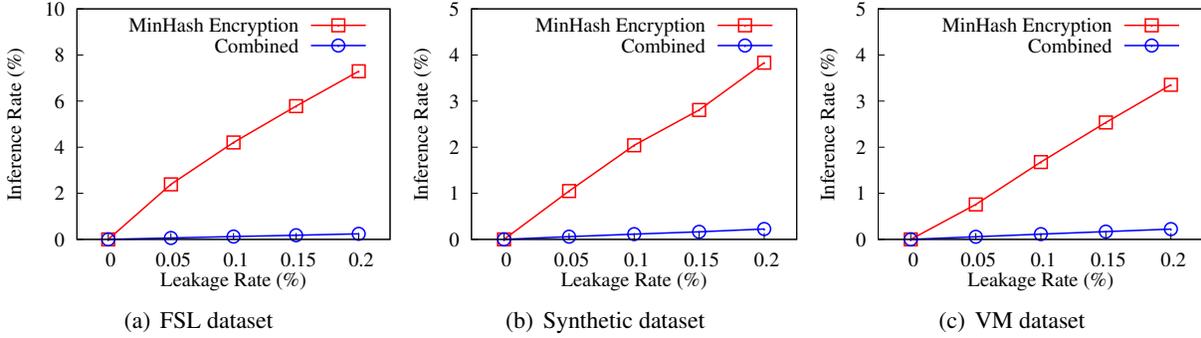
Figure 10: Defense effectiveness: Inference rate in known-plaintext mode under MinHash encryption only and the combined MinHash encryption and scrambling scheme.

(a) FSL dataset   (b) Synthetic dataset   (c) VM dataset
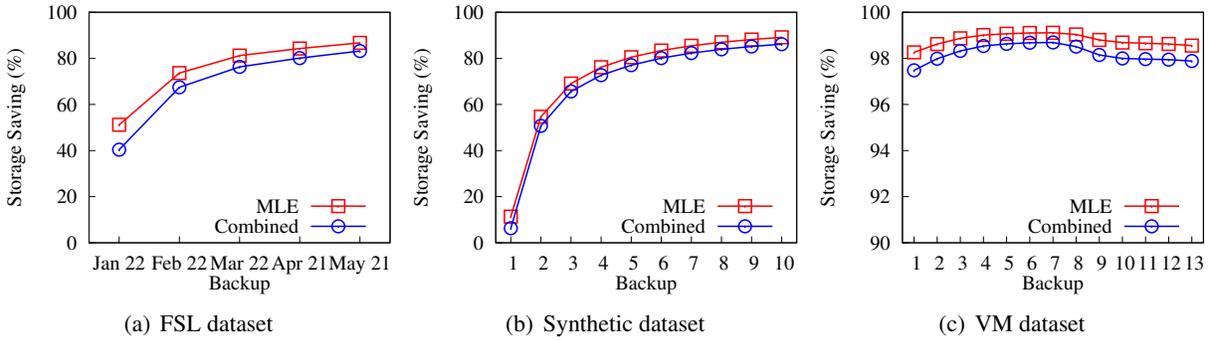
Figure 11: Storage efficiency of the combined MinHash encryption and scrambling scheme.

7.2 Defense Effectiveness

We evaluate our defense schemes, including (i) MinHash encryption only and (ii) the combined MinHash encryption and scrambling scheme, against the advanced locality-based attack in known-plaintext mode under the same parameter setting as in Section 5.3. Note that the advanced locality-based attack reduces to the locality-based attack in the VM dataset, which uses fixed-size chunking.

Figure 10 shows the inference rate versus the leakage rate. When the leakage rate is 0.2%, MinHash encryption suppresses the inference rate to 7.3%, 3.8%, and 3.4% for the FSL, synthetic, and VM datasets, respectively, under the advanced locality-based attack. In addition, the combined MinHash encryption and scrambling scheme further suppresses the inference rate to 0.2-0.24% only for all datasets. This shows that scrambling effectively enhances the protection of MinHash encryption, and the combined scheme effectively defends against the advanced locality-based attack.

7.3 Storage Efficiency

We evaluate the storage efficiency of the combined MinHash encryption and scrambling scheme. Specifically, we add the encrypted backups to storage in the order of their creation times, and measure the storage saving as the percentage of the total size of all ciphertext chunks reduced by deduplication. We compare the storage saving with that of the original MLE, which performs chunk-based deduplication that operates at the more fine-grained chunk level and eliminates all duplicate chunks. Here, we do not consider the metadata overhead.

Figure 11(a) shows the storage saving after storing each FSL backup. After storing all five backups, the

combined scheme achieves a storage saving of 83.2% (which corresponds to a deduplication ratio of 6.0×), which is 3.6% less than that of MLE.

Figure 11(b) shows the storage saving after storing each synthetic snapshot. After 11 backups, the combined scheme achieves a storage saving of 86.2% (which corresponds to the deduplication ratio of 7.2×). The drop of the storage saving is about 3% compared to MLE, which achieves a storage saving of 89.2% (which corresponds to a deduplication ratio of 9.3×).

Figure 11(c) shows the storage saving for the VM dataset. Overall, the storage saving for the first backup reaches 97.4% (which corresponds to a deduplication ratio of 38.5×), mainly because the VM images are initially installed with the same operating system. The storage saving drops after the 7th backup, since the students make big changes and add unique chunks into the VM images. After 13 backups, the storage saving of the combined scheme achieves 97.9% (which corresponds to a deduplication ratio of 47.6×), with a reduction of 0.7% compared to that of MLE.

Overall, the combined scheme maintains high storage efficiency achieved by deduplication for all datasets.

7.4 Metadata Access Overhead

We evaluate the performance of the combined MinHash encryption and scrambling scheme via a case study of its deployment. We implement a deduplication prototype based on the Data Domain File System (DDFS) [67] to simulate the processing of encrypted deduplication workload. DDFS has been used in production backup management for over 15 years [5, 23]. Its chunk locality design also lays the foundation of various follow-up deduplicated storage systems (e.g., [45, 47, 65]). Thus, we believe that DDFS is representative and our study results for DDFS can also be applied to other locality-based deduplicated storage systems.

Suppose that the chunks have been encrypted, by either the original MLE-based deterministic encryption or our combined MinHash encryption and scrambling scheme. We focus on the metadata access overhead under our DDFS-like prototype, since metadata access plays an important role in deduplication performance [67].

7.4.1 Prototype Design

We design and implement our deduplication prototype based on DDFS. Specifically, our prototype organizes the unique (ciphertext) chunks on disk in units of *containers* [44]. Each container size is typically of several megabytes (e.g., 4 MB) to mitigate the disk seek overhead, as opposed to the chunk size that is often of several kilobytes (e.g., 4 KB or 8 KB). In addition, our prototype maintains a *fingerprint index* to hold the metadata (e.g., the mappings of fingerprints to chunk locations) and detect if any identical chunk has been stored. Since the size of the fingerprint index increases with the amount of unique chunks being stored, the fingerprint index is stored on disk, while our prototype maintains two in-memory data structures, namely a *fingerprint cache* and a *Bloom filter*, to mitigate the disk I/O overhead during deduplication (see below).

Our prototype follows the deduplication workflow of DDFS [67]. In particular, it stores unique chunks in logical order and further exploits chunk locality to accelerate deduplication. Given an incoming ciphertext chunk C, our prototype performs deduplication as follows.

- Step S1: Our prototype checks by fingerprint if C is in the fingerprint cache. If so, it is identical and does not need to be stored.

- Step S2: If C is not in the fingerprint cache, our prototype checks the Bloom filter. If C is not in the Bloom filter, it must be unique. Then our prototype updates the Bloom filter, and also inserts C and its fingerprint into an in-memory fixed-size buffer in logical order. If the in-memory buffer is full, our prototype flushes it to disk as a new container and updates the fingerprint index on disk.
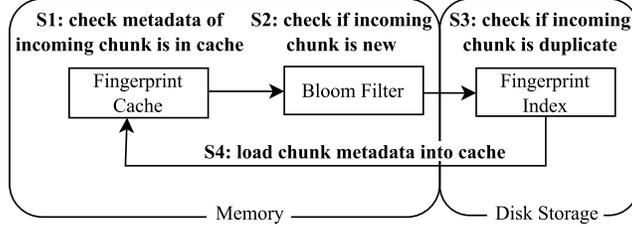
Figure 12: Metadata flow of our deduplication prototype.

- Step S3: Even if C is in the Bloom filter, it may be a false positive. Our prototype queries the fingerprint index to ensure that it is a duplicate. If C is not in the fingerprint index, our prototype follows Step S2 to store C as a unique chunk.

- Step S4: If C is in the fingerprint index, our prototype identifies the container that keeps the physical copy of C, and *loads the fingerprints of all chunks in the container* into the fingerprint cache. The rationale is that the logically nearby chunks of C are likely to be accessed together due to chunk locality. If the fingerprint cache is full, our prototype removes the least-recently-used fingerprints.

Our prototype mainly implements the metadata flow during deduplication, as shown in Figure 12. We focus on the evaluation of the metadata access overhead. We do not evaluate the performance of writing or reading containers and that of encrypting or decrypting chunks.

7.4.2 Evaluation Results

Our evaluation uses the following configurations. Here, we only focus on the FSL dataset. We set the metadata size of each fingerprint as 32 bytes. We consider two sizes of the fingerprint cache: 512 MB and 4 GB. We set the Bloom filter with a false positive rate of 0.01 [67], and the Bloom filter size depends on the number of fingerprints that are tracked. For example, our FSL dataset contains around 65 million fingerprints (i.e., the total size is around 2 GB), so we use 7 hash functions and the corresponding Bloom filter size is around 74 MB. We also set the container size as 4 MB. For the combined scheme, we configure its minimum, average and maximum segment sizes as 512 KB, 1 MB and 2 MB, respectively.

We categorize the on-disk metadata access into three types: (i) *update access*, which updates the metadata of unique chunks in the fingerprint index (in Steps S2 and S3); (ii) *index access*, which looks up the on-disk fingerprint index for the detection of duplicate chunks (in Step S3); and (iii) *loading access*, which loads the fingerprints of stored chunks into the cache (in Step S4). We measure the metadata access overhead in terms of the size of metadata being accessed.

In the following, we compare the metadata access overhead of our combined MinHash encryption and scrambling scheme with MLE, in which we encrypt the chunks by the original MLE-based deterministic encryption.

Figure 13 first presents the results when the fingerprint cache size is 512 MB, in which case the size is insufficient to hold all fingerprints in the FSL dataset (whose total metadata size for all fingerprints is around 2 GB). Figure 13(a) shows the overall metadata access overhead. In the first backup, the combined scheme even incurs less metadata access overhead than MLE, mainly because it generates more unique chunks at the beginning and reduces the frequency of loading fingerprints from disk into the fingerprint cache (in Step S4). In the subsequent backups, the combined scheme has slightly higher overhead than MLE (at most 1.2%), since it generates more unique chunks and needs to load fingerprints more often from disk to the fingerprint cache. Figures 13(b) and 13(c) show the breakdown of the metadata access overhead for MLE and the combined scheme, respectively. The update access size for both schemes is less than 0.3 GB after the first

(a) Overall metadata access overhead   (b) Breakdown for MLE   (c) Breakdown for combined
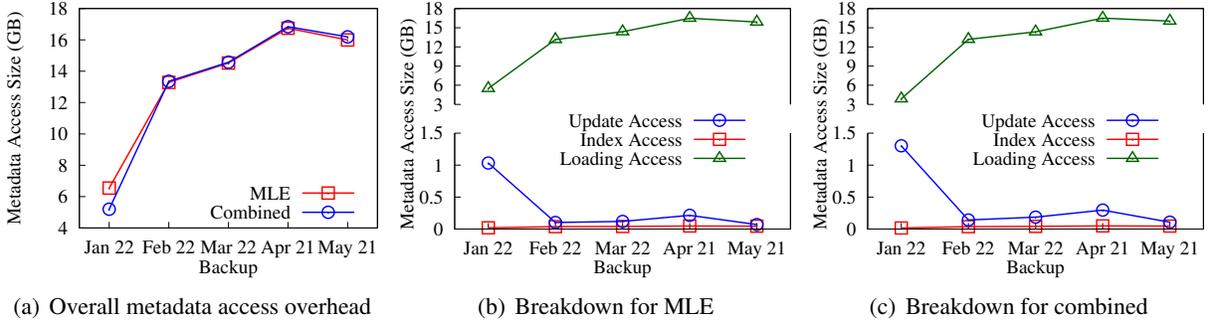
Figure 13: Performance evaluation: Metadata access overhead when the fingerprint cache size is 512 MB.

(a) Overall metadata access overhead   (b) Breakdown for MLE   (c) Breakdown for combined
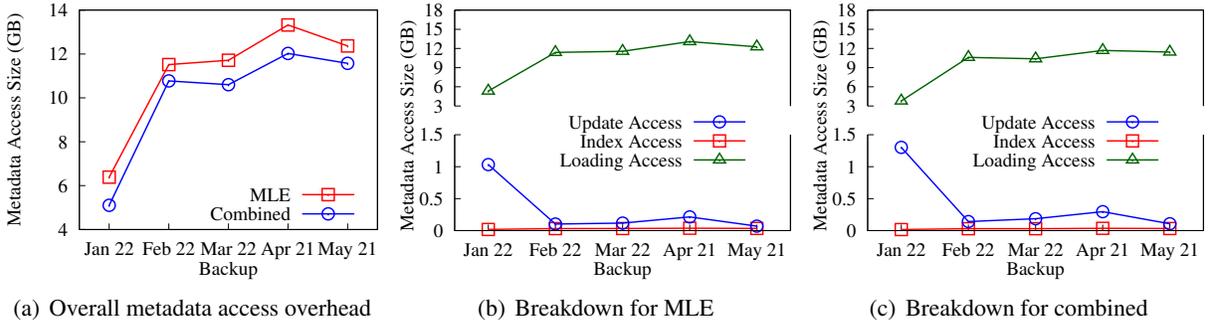
Figure 14: Performance evaluation: Metadata access overhead when the fingerprint cache size is 4 GB.

backup (in which MLE and the combined scheme incur 1.0 GB and 1.3 GB of metadata access, respectively), as only a small portion of new or modified chunks are stored. The index access size is also small, with less than 0.1 GB for both schemes in all backups, since a significant portion of duplicate and unique chunks can be detected by the fingerprint cache and the Bloom filter, respectively. Finally, we observe that the loading access size contributes the most overhead, with more than 74.2% of the total metadata access size for both schemes.

Figure 14 presents the results when the fingerprint cache size is increased to 4 GB, in which the fingerprint cache is sufficiently large to hold the fingerprints of all unique chunks. As shown in Figure 14(a), the combined scheme incurs much less metadata access overhead than MLE by 6.4-20.0% as it generates more unique chunks while all fingerprints can be stored in the fingerprint cache. Figures 14(b) and 14(c) show the corresponding breakdown for MLE and the combined scheme, respectively. Both update access size and index access size are similar to those in Figure 13, while the loading access size for both schemes is significantly reduced by around 22% and 29% for MLE and the combined schemes, respectively, mainly due to a high probability of cache hits.

8   Related Work

Optimizing deduplication: Existing deduplication studies (see a complete survey [64] on deduplication) exploit workload characteristics (e.g., chunk locality [38, 45, 65, 67] and file similarity [14, 65]) to mitigate indexing overhead. For example, DDFS [67] prefetches the fingerprints of nearby chunks that are likely to be accessed together. Sparse Indexing [45] and Extreme Binning [14] exploit chunk locality and file similarity, respectively, to mitigate the memory storage for indexing, while SiLo [65] combines both chunk locality and file similarity for general backup workloads. Bimodel [38] builds on chunk locality and adaptively varies

the expected chunk sizes to mitigate metadata overhead. All the above works do not consider security.

Encrypted deduplication: Traditional encrypted deduplication systems (e.g., [7, 20, 22, 35, 58, 63]) mainly build on convergent encryption [22], in which the encryption key is directly derived from the cryptographic hash of the content to be encrypted. CDStore [43] integrates convergent encryption with secret sharing to support fault-tolerant storage. Metadedup [41] extends CDStore with space-efficient metadata management. However, convergent encryption is vulnerable to brute-force attacks (see Section 2.2). Server-aided MLE protects against brute-force attacks by maintaining content-to-key mappings in a dedicated key manager, and has been implemented in various storage system prototypes [8, 12, 53, 56]. Given that the dedicated key manager is a single-point-of-failure, Duan [24] proposes to maintain a quorum of key managers via threshold signature for fault-tolerant key management. Note that all the above systems build on deterministic encryption to preserve the deduplication capability of ciphertext chunks, and hence are vulnerable to the inference attacks studied in this paper.

Instead of using deterministic encryption, Bellare *et al.* [13] propose an MLE variant called *random convergent encryption (RCE)*, which uses random keys for chunk encryption. However, RCE needs to add deterministic tags into ciphertext chunks for checking any duplicates, so that the adversary can count the deterministic tags to obtain the frequency distribution. Liu *et al.* [46] propose to encrypt each plaintext chunk with a random key, while the key is shared among users via password-based key exchange. However, the proposed approach incurs significant key exchange overhead, especially when the number of chunks is huge.

From the theoretic perspective, several studies propose to enhance the security of encrypted deduplication and protect the frequency distribution of original chunks. Abadi *et al.* [3] propose two encrypted deduplication schemes for the chunks that depend on public parameters, yet either of them builds on computationally expensive non-interactive zero knowledge (NIZK) proofs or produces deterministic ciphertext components. Interactive MLE [11] addresses chunk correlation and parameter dependence, yet it is impractical for the use of fully homomorphic encryption (FHE). This paper differs from the above works by using lightweight primitives for practical encrypted deduplication.

Inference attacks: Frequency analysis [48] is the classical inference attack and has been historically used to recover plaintexts from substitution-based ciphertexts. It is also used as a building block in recently proposed attacks. Kumar *et al.* [39] use frequency-based analysis to de-anonymize query logs. Islam *et al.* [33] compromise keyword privacy based on the leakage of the access patterns in keyword search. Naveed *et al.* [50] propose to conduct frequency analysis via combinatorial optimization and present attacks against CryptDB. Kellaris *et al.* [36] propose reconstruction attacks against any system that leaks access pattern or communication volume. Pouliot *et al.* [52] present the graph matching attacks on searchable encryption. Grubbs *et al.* [28] build attacks on order-preserving encryption based on the frequency and ordering information.

In encrypted deduplication, Ritzdorf *et al.* [55] exploit the size information of deduplicated content and build an inference attack that determines if a file has been stored. Armknecht *et al.* [9] present formal analysis on the side-channel attack that just works in client-side deduplication. Our work is different as we focus on inferring the content of data chunks via frequency analysis. In particular, we exploit workload characteristics to construct attack and defense approaches.

Some inference attacks exploit the *active* adversarial capability. Brekne *et al.* [17] construct bogus packets to de-anonymize IP addresses. Cash *et al.* [19] and Zhang *et al.* [66] propose file-injection attacks against searchable encryption. Our proposed attacks do not rely on the active adversarial capability.

9 Conclusion

Encrypted deduplication has been deployed in commercial cloud environments and extensively studied in the literature to simultaneously achieve both data confidentiality and storage efficiency, yet we argue that its data confidentiality remains not fully guaranteed. We demonstrate how the deterministic nature of encrypted deduplication makes it susceptible to information leakage caused by frequency analysis. We propose the locality-based attack, which exploits the chunk locality property of backup workloads to infer the content of a large fraction of plaintext chunks from the ciphertext chunks of the latest backup. We also propose the advanced locality-based attack, which extends the locality-based attack with the knowledge of chunk sizes to launch frequency analysis specifically against variable-size chunks. We show how the inference attacks can be practically implemented, and demonstrate their severities through trace-driven evaluation on both real-world and synthetic datasets. To defend against information leakage, we consider MinHash encryption and scrambling to disturb frequency rank and break chunk locality. Our trace-driven evaluation shows that our combined MinHash encryption and scrambling effectively defends against the locality-based attack, while maintaining high storage efficiency and incurring limited metadata access overhead.

References

[1] FSL traces and snapshots public archive. http://tracer.filesystems.org/, 2014.

[2] Ubuntu irc logs. http://irclogs.ubuntu.com, 2019.

[3] Martín Abadi, Dan Boneh, Ilya Mironov, Ananth Raghunathan, and Gil Segev. Message-locked encryption for lock-dependent messages. In *Advances in Cryptology – CRYPTO 2013*, pages 374–391, 2013.

[4] Ibrahim A. Al-Kadit. Origins of Cryptology: The Arab Contributions. *Cryptologia*, 16(2):97–126, 1992.

[5] Yamini Allu, Fred Douglis, Mahesh Kamat, Philip Shilane, Hugo Patterson, and Ben Zhu. Backup to the future: How workload and hardware changes continually redefine data domain file systems. *IEEE Trans. Comput.*, 50(7):64–72, 2017.

[6] George Amvrosiadis and Medha Bhadkamkar. Identifying trends in enterprise data protection systems. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC'15)*, 2015.

[7] Paul Anderson and Le Zhang. Fast and secure laptop backups with encrypted de-duplication. In *Proceedings of the 24th International Conference on Large Installation System Administration (LISA'10)*, pages 1–8, 2010.

[8] Frederik Armknecht, Jens-Matthias Bohli, Ghassan O. Karame, and Franck Youssef. Transparent data deduplication in the cloud. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*, pages 886–900, 2015.

[9] Frederik Armknecht, Colin Boyd, Gareth T. Davies, Kristian Gjøsteen, and Mohsen Toorani. Side channels in deduplication: Trade-offs between leakage and efficiency. In *Proceedings of ACM Asia Conference on Computer and Communications Security (ASIACCS'17)*, pages 266–274, 2017.

[10] Michael Arrington. AOL: "this was a screw up". https://techcrunch.com/2006/08/07/aol-this-was-a-screw-up/, 2006.

[11] Mihir Bellare and Sriram Keelveedhi. Interactive message-locked encryption and secure deduplication. In *Public-Key Cryptography – PKC 2015*, pages 516–538, 2015.

[12] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. DupLESS: Server-aided encryption for deduplicated storage. In *Proceeding of the 22nd USENIX Security Symposium (USENIX Security'13)*, pages 179–194, 2013.

[13] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Message-locked encryption and secure deduplication. In *Advances in Cryptology – EUROCRYPT 2013*, pages 296–312, 2013.

[14] Deepavali Bhagwat, Kave Eshghi, Darrell D.E. Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceeding of IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS'09)*, pages 1–9, 2009.

[15] Vincent Bindschaedler, Paul Grubbs, David Cash, Thomas Ristenpart, and Vitaly Shmatikov. The tao of inference in privacy-protected databases. In *Proceedings of the VLDB Endowment*, volume 11, pages 1715–1728, 2018.

[16] John Black. Compare-by-hash: a reasoned analysis. In *Proceeding of USENIX Annual Technical Conference (USENIX ATC'06)*, pages 85–90, 2006.

[17] Tønnes Brekne, André Årnes, and Arne Øslebø. Anonymization of IP traffic monitoring data: Attacks on two prefix-preserving anonymization schemes and some proposed remedies. In *Proceeding of International Workshop on Privacy Enhancing Technologies (PET'05)*, pages 179–196, 2005.

[18] Andrei Z. Broder. On the resemblance and containment of documents. In *Proceeding of the Compression and Complexity of Sequences (SEQUENCES'97)*, pages 21–29, 1997.

[19] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*, pages 668–679, 2015.

[20] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, pages 285–298, 2002.

[21] Barb Darrow. Harvard-affiliate McLean hospital loses patient data. `http://fortune.com/2015/07/29/mclean-hospital-loses-patient-data/`, 2015.

[22] 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 *Proceeding of 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 617–624, 2002.

[23] Fred Douglis, Abhinav Duggal, Philip Shilane, Tony Wong, Shiqin Yan, and Fabiano Botelho. The logic of physical garbage collection in deduplicating storage. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 29–43, 2017.

[24] Yitao Duan. Distributed key generation for encrypted deduplication: Achieving the strongest privacy. In *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security (CCSW'14)*, pages 57–68, 2014.

[25] Kave Eshghi and Hsiu Khuern Tang. A framework for analyzing and improving content-based chunking algorithms. HPL-2005-30R1, 2005.

[26] Sanjay Ghemawat and Jeff Dean. LevelDB: A fast key/value storage library by Google. https://github.com/google/leveldb, 2014.

[27] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *Proceedings of ACM Conference on Computer and Communications Security (CCS'16)*, pages 1353–1364, 2016.

[28] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *Proceeding of IEEE Symposium on Security and Privacy (SP'17)*, pages 655–672, 2017.

[29] Robert Hackett. Linkedin lost 167 million account credentials in data breach. `http://fortune.com/2016/05/18/linkedin-data-breach-email-password/`, 2016.

[30] Shai Halevi, Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Proofs of ownership in remote storage systems. In *Proceedings of the 18th ACM conference on Computer and Communications Security (CCS'11)*, pages 491–500, 2011.

[31] 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.

[32] HIPAA Journal. Hard drive theft sees data of 1 million individuals exposed. https://www.hipaajournal.com/hard-drive-theft-sees-data-1-million-individuals-exposed-8859/, 2017.

[33] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Proceeding of Network and Distributed System Security Symposium (NDSS'12)*, pages 1–15, 2012.

[34] Keren Jin and Ethan L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceeding of the Israeli Experimental Systems Conference (SYSTOR'09)*, pages 7:1–7:12, 2009.

[35] Mahesh Kallahall, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of USENIX Conference on File and Stroage Technologies (FAST'03)*, pages 29–42, 2003.

[36] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *Proceedings of ACM Conference on Computer and Communications Security (CCS'16)*, pages 1329–1340, 2016.

[37] Bryan Klimt and Yiming Yang. The enron corpus: A new dataset for email classification research. In *Proceeding of European Conference on Machine Learning*, pages 217–226, 2004.

[38] Erik Kruus, Cristian Ungureanu, and Cezary Dubnicki. Bimodal content defined chunking for backup streams. In *Proceeding of USENIX Conference on File and Storage Technologies (FAST'10)*, 2010.

[39] Ravi Kumar, Jasmine Novak, Bo Pang, and Andrew Tomkins. On anonymizing query logs via token-based hashing. In *Proceedings of the 16th international conference on World Wide Web (WWW'07)*, pages 629–638, 2007.

[40] Marie-Sarah Lacharité and Kenneth G. Paterson. A note on the optimality of frequency analysis vs. l_p-optimization. Cryptology ePrint Archive: Report 2015/1158 `https://eprint.iacr.org/2015/1158`, 2015.

[41] Jingwei Li, Patrick P. C. Lee, Yanjing Ren, and Xiaosong Zhang. Metadedup: Deduplicating metadata in encrypted deduplication via indirection. In *Proceeding of the 35th International Conference on Massive Storage Systems and Technology (MSST'19)*, pages 1–13, 2019.

[42] Jingwei Li, Chuan Qin, Patrick P. C. Lee, and Xiaosong Zhang. Information leakage in encrypted deduplication via frequency analysis. In *Proceeding of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'17)*, pages 1–12, 2017.

[43] Mingqiang Li, Chuan Qin, and Patrick P. C. Lee. CDStore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC'15)*, pages 111–124, 2015.

[44] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proceeding of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, pages 183–197, 2013.

[45] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceeding of USENIX Conference on File and Storage Technologies (FAST'09)*, pages 111–123, 2009.

[46] Jian Liu, N. Asokan, and Benny Pinkas. Secure deduplication of encrypted data without additional independent servers. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*, pages 874–885, 2015.

[47] Jingwei Ma, Rebecca J. Stones, Yuxiang Ma, Jingui Wang, Junjie Ren, Gang Wang, and Xiaoguang Liu. Lazy exact deduplication. In *Proceeding of 32nd Symposium on Mass Storage Systems and Technologies (MSST'16)*, pages 1–10, 2016.

[48] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[49] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*, pages 1–1, 2011.

[50] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference attacks on property-preserving encrypted databases. In *Proceeding of 22nd ACM Conference on Computer and Communications Security (CCS'15)*, pages 644–655, 2015.

[51] Muhammad Naveed, Manoj Prabhakaran, and Carl A. Gunter. Dynamic searchable encryption via blind storage. In *Proceedings of IEEE Symposium on Security and Privacy (SP'14)*, pages 639–654, May 2014.

[52] David Pouliot and Charles V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *Proceedings of the 23th ACM Conference on Computer and Communications Security (CCS'16)*, pages 1341–1352, 2016.

[53] Chuan Qin, Jingwei Li, and Patrick P. C. Lee. The design and implementation of a rekeying-aware encrypted deduplication storage system. *ACM Trans. on Storage*, 13(1):9:1–9:30, Mar 2017.

[54] Michael O. Rabin. Fingerprinting by random polynomials. Center for Research in Computing Technology, Harvard University. Tech. Report TR-CSE-03-01, 1981.

[55] Hubert Ritzdorf, Ghassan Karame, Claudio Soriente, and Srdjan Čapkun. On information leakage in deduplicated storage systems. In *Proceedings of ACM on Cloud Computing Security Workshop (CCSW'16)*, pages 61–72, 2016.

[56] Peter Shah and Won So. Lamassu: Storage-efficient host-side encryption. In *Proceedings of USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'15)*, pages 333–345, 2015.

[57] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $o((\log n)^3)$ worst-case cost. In *Advances in Cryptology – ASIACRYPT 2011*, pages 197–214, 2011.

[58] Mark W. Storer, Kevin Greenan, Darrell D.E. Long, and Ethan L. Miller. Secure data deduplication. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability (StorageSS'08)*, pages 1–10, 2008.

[59] Zhu Sun, Geoff Kuenning, Sonam Mandal, Philip Shilane, Vasily Tarasov, Nong Xiao, and Erez Zadok. A long-term user-centric analysis of deduplication patterns. In *Proceeding of the 32nd Symposium on Mass Storage Systems and Technologies (MSST'16)*, 2016.

[60] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. Generating realistic datasets for deduplication analysis. In *Proceedings of USENIX conference on Annual Technical Conference (USENIX ATC'12)*, pages 24–24, 2012.

[61] David C. Uthus and David W. Aha. The ubuntu chat corpus for multiparticipant chat analysis. In *Proceeding of AAAI Spring Symposium*, pages 99–102, 2013.

[62] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST'12)*, pages 33–48, 2012.

[63] Zooko Wilcox-O'Hearn and Brian Warner. Tahoe: The least-authority filesystem. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability (StorageSS'08)*, pages 21–26, 2008.

[64] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[65] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. SiLo: A similarity locality based near exact deduplication scheme with low ram overhead and high throughput. In *Proceeding of USENIX Annual Technical Conference (USENIX ATC'11)*, pages 285–298, 2011.

[66] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: the power of file-injection attacks on searchable encryption. In *Proceeding of the 25th USENIX Security Symposium (Security'16)*, pages 707–720, 2016.

[67] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, pages 269–282, 2008.