FeatureSpy: Detecting Learning-Content Attacks via Feature Inspection in Secure Deduplicated Storage

Jingwei Li[†], Yanjing Ren[‡], Patrick P. C. Lee[‡], Yuyu Wang[†], Ting Chen[†], and Xiaosong Zhang[†]
[†]*University of Electronic Science and Technology of China* [‡]*The Chinese University of Hong Kong*

***Abstract*—Secure deduplicated storage is a critical paradigm for cloud storage outsourcing to achieve both operational cost savings (via deduplication) and outsourced data confidentiality (via encryption). However, existing secure deduplicated storage designs are vulnerable to learning-content attacks, in which malicious clients can infer the sensitive contents of outsourced data by monitoring the deduplication pattern. We show via a simple case study that learning-content attacks are indeed feasible and can infer sensitive information in short time under a real cloud setting. To this end, we present FeatureSpy, a secure deduplicated storage system that effectively detects learning-content attacks based on the observation that such attacks often generate a large volume of similar data. FeatureSpy builds on two core design elements, namely (i) similarity-preserving encryption that supports similarity detection on encrypted chunks and (ii) shielded attack detection that leverages Intel SGX to accurately detect learning-content attacks without being readily evaded by adversaries. Trace-driven experiments on real-world and synthetic datasets show that our FeatureSpy prototype achieves high accuracy and low performance overhead in attack detection.**

I. Introduction

Enterprise and individual clients often outsource storage management to the cloud, so as to save the costs of self-managing huge amounts of data. To make storage outsourcing cost-efficient, *source-based deduplication* can be used to eliminate the transfers and storage of duplicate contents from a client to the cloud [26]. Specifically, before the client uploads a data chunk to the cloud, it first computes the *fingerprint* (e.g., cryptographic hash) of the data chunk. It sends the fingerprint to the cloud, which checks if the same data chunk copy with the same fingerprint has already been stored (by the client itself or other clients). If so, the client does not need to upload the data, thereby achieving both bandwidth and storage savings.

Cost-efficient storage outsourcing should provide security guarantees against malicious attacks by the cloud or even clients themselves. There are two known secure approaches (elaborated in §II-A): (i) *encrypted deduplication* [18], [19], in which clients encrypt data chunks before source-based deduplication for confidentiality, while ensuring that duplicate data chunks are always encrypted to the same encrypted chunks to maintain deduplication effectiveness; and (ii) *proof-of-ownership (PoW)* [25], which requires a client to prove to the cloud that it indeed has authorized access to its outsourced data chunks, so as to protect against *side-channel attacks* [25], [26], [39] that allow malicious clients to obtain unauthorized access to the outsourced data chunks.

Corresponding author: Ting Chen (brokendragon@uestc.edu.cn)

However, existing secure deduplicated storage designs, even coupled with encrypted deduplication and PoW, remain vulnerable to *learning-content attacks* [26], [57], in which a malicious client can upload multiple forged files and check if the upload of any forged file is eliminated due to source-based deduplication; if so, it implies that the forged file is duplicate with some currently stored file. Learning-content attacks are indeed practical and severe, as an adversary can infer sensitive content (e.g., salary in an employment offer letter) within a short time period (e.g., a few minutes as shown in §II-B). Note that encrypted deduplication and PoW cannot defend against learning-content attacks, since the malicious clients only intentionally craft the forged files, yet they still follow the same upload protocol as benign clients.

We present FeatureSpy, a secure deduplicated storage system that augments encrypted deduplication and PoW with the detection of learning-content attacks. FeatureSpy builds on the insight that in learning-content attacks, a malicious client needs to generate a sufficient number of *similar* (but non-duplicate) data chunks for the attack to be effective; in contrast, practical (non-compromised) workloads are less likely to have many similar data chunks in a short time period. Based on the insight, FeatureSpy examines the content similarity across data chunks and reports the attack if many similar data chunks are found. To summarize, this paper makes the following contributions:

- We show via a case study that a malicious client can feasibly launch learning-content attacks in the LAN and cloud environments in a short time period (e.g., two and eight minutes, respectively).
- We design FeatureSpy to support secure deduplicated storage with the detection of learning-content attacks. One key challenge of the design is that a malicious client can tamper with the attack detection procedure. To this end, we design FeatureSpy with two new primitives: (i) *similarity-preserving encryption*, which preserves content similarity in encryption, so that attack detection can be applied to encrypted data chunks; and (ii) *shielded attack detection*, which provides shielded executions for attack detection via Intel SGX [5], in order to prevent evasion from malicious clients.
- We conduct extensive trace-driven experiments using real-world and synthetic datasets. We show that FeatureSpy is configurable about the trade-off between the detection accuracy and false positive rate in detecting learning-content attacks. In the default configuration, even the forged files have limited similarity (e.g., 3.1% contents are modified),

it preserves the detection accuracy of at least 73% with the false positive rate of up to 11%. FeatureSpy also incurs as low as 15.0% and 0.8% throughput drops in uploads and downloads, respectively, compared with SGXDedup [45], a state-of-the-art high-performance secure deduplicated storage system that cannot defend against learning-content attacks.

The source code of FeatureSpy prototype is available at **https://github.com/tinoryj/FeatureSpy**.

II. Background and Motivation

A. Basics

Deduplication. Deduplication is a well-studied redundancy elimination technique being widely deployed in modern storage management. It is shown to effectively achieve substantial storage savings in practical workloads such as virtual disk images [28], file systems [36], backups [52], and container images [56]. It works by partitioning data into fixed-size or variable-size *chunks*. Each chunk is identified by the cryptographic hash (i.e., *fingerprint*) of the corresponding content, assuming that it is highly unlikely to have distinct chunks with the same fingerprint [20]. Deduplication uses a key-value store (called the *fingerprint index*) that tracks the fingerprints of all already stored chunks. It only stores a new chunk whose fingerprint is new to the fingerprint index, thereby achieving storage savings.

In this paper, we focus on applying deduplication into outsourced storage, in which a client stores data in a third-party storage service provider (e.g., cloud). The cloud maintains the fingerprint index and performs (global) deduplication on the data from the same or different clients. We also consider source-based deduplication [26], in which a client uploads only the non-duplicate chunks (§I) to the cloud to save both bandwidth and storage costs of duplicate chunks.

Encrypted deduplication. To protect outsourced data from being eavesdropped by external adversaries and the cloud, we consider *encrypted deduplication*, which encrypts input *plaintext* chunks into *ciphertext* chunks via symmetric-key encryption before deduplication, while preserving the deduplication effectiveness on ciphertext chunks. The core idea of encrypted deduplication is to derive the key from the chunk content, so that duplicate plaintext chunks are always encrypted into duplicate ciphertext chunks. *Message-locked encryption (MLE)* [19] is a cryptographic primitive that formalizes how the symmetric key (called the *MLE key*) is derived from the chunk content. One MLE implementation is *convergent encryption* [23], which computes the MLE key as the cryptographic hash of the plaintext chunk content. Another MLE implementation is *server-aided MLE* [18], which performs MLE key generation in a dedicated key manager in order to resist the offline brute-force attacks against convergent encryption. Our work builds on server-aided MLE.

Proof-of-ownership (PoW). In source-based encrypted deduplication, a client first submits the fingerprint of a ciphertext chunk to the cloud, and uploads only the ciphertext chunk that is deemed non-duplicate. However, source-based encrypted deduplication is vulnerable to *side-channel attacks* [25], [26], [39]. One side-channel attack is that a malicious client can submit the fingerprint of some target ciphertext chunk to check if the chunk has already been stored [26]. Another side-channel attack [25], [39] is that a malicious client can use the fingerprint of some target ciphertext chunk to mislead the cloud that it has full access to the chunk, thereby obtaining unauthorized rights for downloading the chunk [25], [39].

Source-based encrypted deduplication can be coupled with *proof-of-ownership (PoW)* [25] to defend against the above side-channel attacks. Specifically, when the client submits the fingerprint of a ciphertext chunk to the cloud, it also attaches a short verification value (called the *proof*) generated by a PoW protocol between the client and the cloud. Based on the proof, the cloud verifies if the client is the owner that actually holds the ciphertext chunk in entirety.

B. Learning-Content Attacks

Secure deduplicated storage, even under the protection of encrypted deduplication and PoW, remains vulnerable to *learning-content attacks* [26], [57]. In such attacks, we assume that (i) a malicious client knows a priori the most of the content of a target file of some victim client and (ii) the message space of the remaining sensitive content has a limited number of possibilities. Then the malicious client aims to infer the remaining sensitive content of the target file by (i) creating forged files with different possibilities of the sensitive content, (ii) performing source-based deduplication on each forged file, and (iii) deducing the forged file as the target file if no chunks of the forged file need to be uploaded (i.e., the forged file is duplicate in entirety). Since each forged file is created and owned by the malicious client, PoW is insufficient to defend against learning-content attacks.

While learning-content attacks are known [26], [57], it remains open whether such attacks are feasible in practice. We show via a case study that learning-content attacks are indeed feasible and severe. Suppose that Alice received an employment offer letter, and Trudy wants to infer Alice's salary in her offer. Both Alice and Trudy belong to the same organization that manages backups in an outsourced storage system with source-based deduplication, protected under encrypted deduplication and PoW. Alice stores her offer letter as a backup, and Trudy's goal is to create different forged letters, with all possible salaries, to infer if any forged letter matches Alice's offer letter.

To simulate the attack scenario, we start with Google's offer letter template [8]. We change the *Name*, *annual salary* (say, a multiple of 6 K [26] between 204 K and 804 K), and *sign-on bonus* (say, a multiple of 10 K between 300 K and 600 K) to generate the offer letters, each of which has a size of 18.5 KiB. In source-based deduplication, suppose that each letter file is partitioned into chunks via Rabin fingerprinting [43] with an average chunk size of 8 KiB [50]. Each plaintext chunk is encrypted under MLE, and its ciphertext chunk is uploaded only if it is non-duplicate. Initially, Alice first uploads her offer letter. Trudy creates forged letters with different possible annual salary and sign-on bonus amounts. He uploads each forged

Setups	**# uploads**	**Traffic**	**Time**
LAN	841.0 ± 608.3	7.1 ± 5.1 MiB	105.0 ± 76.1 s
Cloud			475.5 ± 339.8 s

TABLE I: Average costs of learning-content attacks. The results are averaged over 10 runs, with 95% confidence intervals under student's t-distribution.

letter, and checks if the full forged letter (if it is non-duplicate), or only metadata information (if it is duplicate), is sent to the cloud by monitoring the network traffic. He stops the uploads if he finds that a forged letter is duplicate.

We simulate the attack ten times and evaluate the attack in both LAN and cloud setups; in the LAN setup, the clients and the cloud are in the same local testbed (see §VI-C for the detailed setup), while in the cloud setup, the clients are in a local testbed and the cloud is deployed as a rented virtual machine (VM) in Alibaba Cloud [1]. Table I shows the average costs for Trudy to infer Alice's offer. On average, Trudy only needs to issue 841 forged letters, or equivalently 7.1 MiB of network traffic (including the transfers of non-duplicate ciphertext chunks and metadata). It takes only 105.0 s and 475.5 s for Trudy to infer Alice's offer letter in the LAN and cloud setups, respectively.

C. *Trusted Execution*

To defend against learning-content attacks in secure deduplicated storage while preserving the bandwidth and storage savings of source-based deduplication, we design a robust mechanism that detects learning-content attacks based on trusted execution technologies. In this work, we choose *Intel Software Guarded Extensions (SGX)* [5] to realize trusted execution, due to its limited performance overhead [27], [45].

SGX extends Intel CPU with security-related instructions inside the CPU hardware. It allocates an *enclave*, which provides a trusted execution environment, in a hardware-guarded memory region (called the *enclave page cache (EPC)*), such that the enclave hosts sensitive contents with confidentiality and integrity guarantees.

Although Intel has deprecated SGX in the next Intel Core platforms [51], it will continuously support SGX in future Intel Xeon platforms [44]. Also, our design has a small trusted computing base (§III), and can be adapted to other trusted execution technology (e.g., ARM TrustZone [40]).

III. FeatureSpy Overview

This paper presents FeatureSpy, a secure deduplicated storage system that augments encrypted deduplication and PoW to detect learning-content attacks. It aims for the following goals:

- **Bandwidth and storage savings.** It maintains bandwidth and storage savings via source-based deduplication.
- **Secure deduplication.** It provides confidentiality for outsourced data via encrypted deduplication and protects source-based deduplication with PoW.
- **Robust and efficient attack detection.** It leverages SGX to reliably detect learning-content attacks against the tampering by some malicious client. It also achieves high accuracy and low misjudgements in attack detection. Furthermore, it performs attack detection on the write path and incurs low performance overhead in SGX.
- **Small trusted computing base.** It manages a small enclave size and minimal function call interfaces for the enclave; this is a necessary design goal to avoid abusing interface function calls [34].

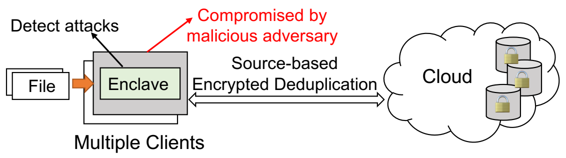
Detect attacks
Compromised by malicious adversary
File
Enclave
Source-based Encrypted Deduplication
Cloud
Multiple Clients

Fig. 1: Architecture of FeatureSpy. An enclave is maintained in each client to detect learning-content attacks.

A. *Architecture*

Figure 1 depicts the architecture of FeatureSpy. We consider a storage outsourcing scenario, in which multiple clients regularly upload data snapshots (e.g., virtual disk images [28], file systems [36], backups [52], and container images [56]) to the cloud for persistent archival storage; such snapshots have high content similarity that favors deduplication (§II-A). A client partitions file data into plaintext chunks and computes the ciphertext chunks and fingerprints for source-based encrypted deduplication.

In FeatureSpy, each client maintains an enclave to detect learning-content attacks. Initially, we compile the enclave code into a shared object [5]. We distribute the shared object, along with a signature, to each client and the cloud for integrity verification. Each client creates the corresponding enclave by loading the shared object, and the cloud authenticates the enclave via *remote attestation* [5] to ensure that the correct code is loaded into the enclave. FeatureSpy allows a benign client to transfer only non-duplicate ciphertext chunks via its enclave to the cloud, and aborts a malicious client that is caught by the enclave for launching learning-content attacks.

B. *Threat Model*

FeatureSpy provides confidentiality guarantees for its outsourced data via encryption against the eavesdropping by external adversaries and the cloud. In addition, FeatureSpy protects against a malicious client that aims to infer the original plaintext chunks of other non-compromised clients. Specifically, the malicious client can access its own plaintext chunks and keys, and launch learning-content attacks by arbitrarily creating forged files. Also, it can tamper with in-memory operations, in order that its attacks can bypass the detection of FeatureSpy. For example, while a benign client processes chunks in multiple phases (e.g., key generation, encryption, and deduplication), the malicious client may skip some phases and inject forged files that are only processed by the remaining phases.

Our threat model makes the following assumptions.

- The communication between each client and the cloud is protected by SSL/TLS against eavesdropping.

- The enclave within each client is trusted and authenticated. It preserves confidentiality for in-enclave contents [45], [48].
- FeatureSpy can protect against corruptions of outsourced data with *proof data possession* [14] and *proof of retrievability* [29], as well as against storage failures via redundancy (e.g., in multi-cloud storage [33]). We do not address such issues in this work.

C. Main Idea and Challenges

Main idea. Our insight is that learning-content attacks often generate many *similar* chunks. By similar, we mean that the chunks are non-duplicate but have largely the same content with information changes in only a few regions. For example, in our case study (§II-B), Trudy enumerates all possible annual salaries and sign-on bonuses, both of which are stored in the same chunk due to the data partitioning of the employment letter template. He forges a large number of chunks that only differ in the salary and bonus amounts. Compared with a chunk of an average size of 8 KiB, the forged chunks only have small content differences (i.e., multiple bytes). Note that our arguments still apply even if the salaries and bonuses are stored in different chunks. In general, we argue that the forged chunks in learning-content attacks cannot have large content differences; otherwise, the attacks will be ineffective by nature due to the extremely high cost of enumerating all possibilities.

From the defense perspective, we observe that practical non-compromised workloads are unlikely to have too many similar chunks that co-occur together. To justify, we analyze the four real-world datasets of backup snapshots, Linux, GCC, CouchDB, and Gitlab (see §VI-A for dataset details). We partition each file of a snapshot into variable-size chunks via Rabin fingerprinting [43] with an average chunk size of 8 KiB [9]. We use the similarity detection scheme [47] (described in §IV-A) to identify similar chunks. Specifically, we transform the Rabin fingerprint of each sliding window of a chunk by multiple linear functions. For each linear function, we generate a *sub-feature* as the Rabin fingerprint of a sliding window whose transformed result has the maximum value across all sliding windows. We derive a *feature* as the concatenation of a number (e.g., four [47]) of sub-features.

Suppose that we generate a single feature for each chunk and find the largest subset of chunks with the same feature within a snapshot. We measure the fraction of chunks of this largest subset over all chunks in the snapshot. Figure 2 shows that the snapshots only have up to 2.1% of chunks (in GCC) with the same feature.

This inspires the design of FeatureSpy, which detects learning-content attacks by examining the similarity of chunks. Specifically, FeatureSpy extracts a set of features of each chunk from a client, and reports the attack if the client generates many chunks with the same set of features.

Challenges. Realizing FeatureSpy in secure deduplicated storage is non-trivial, since the malicious client may tamper with contents and operations (§III-B). In this work, we leverage SGX to run attack detection within the enclave. One major challenge of using SGX is to maintain a small enclave size,

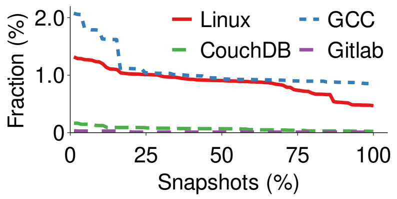

Fig. 2: Fraction of the largest subset of chunks with the same feature over all chunks in a snapshot; the x-axis is the percentile of snapshots, sorted by the fraction in descending order.

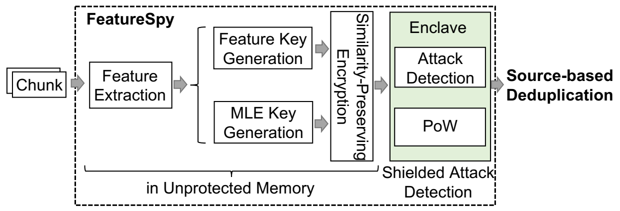

Fig. 3: Design workflow of FeatureSpy.

so as to mitigate performance overhead [13], [22], [27], [45] and security risks [34]. Thus, running all client-side operations inside the enclave is infeasible.

To keep small enclave usage, FeatureSpy opts to run only attack detection and PoW within the enclave, while performing encrypted deduplication (including key management and encryption) in unprotected memory based on server-aided MLE [18] (§II-A). However, this poses a major design issue of whether FeatureSpy should perform attack detection on plaintext chunks (before encryption) or ciphertext chunks (after encryption). For the former, a malicious client can bypass attack detection by injecting forged chunks after the encryption phase (before the chunks are uploaded to the cloud); for the latter, the current encryption operation of MLE (§II-A) destroys content similarity (i.e., similar but non-identical plaintext chunks are encrypted into totally distinct ciphertext chunks) and prohibits our attack detection that builds on the detection of similar chunks.

IV. Detailed Design

In this section, we present the detailed design of FeatureSpy. Figure 3 presents the design workflow. FeatureSpy is deployed in each client, and processes plaintext chunks from the client before they are securely outsourced to the cloud. It first extracts features from each plaintext chunk and derives a feature key (§IV-A). Based on the feature key and MLE key, it performs *similarity-preserving encryption* (§IV-B), which maps similar (but non-identical) and identical plaintext chunks to similar and identical ciphertext chunks, respectively, so as to make our attack detection on ciphertext chunks feasible, while preserving the deduplication effectiveness as in MLE. Finally, it performs *shielded attack detection* (§IV-C), which accurately detects learning-content attacks using small enclave usage, without readily being bypassed by malicious clients.

A. Feature Extraction and Key Generation

We elaborate how FeatureSpy extracts features from the chunk content and performs key generation.

Feature extraction. There are various approaches in the literature (e.g., [21], [47], [55]) to extract features from some data content to capture the content characteristics. In this work, we use the similarity detection scheme in [47]. Note that we do not claim the novelty of this design.

In similarity detection [47], FeatureSpy defines N pairs of coefficients (a_i, m_i), each of which indicates a linear function to transform a Rabin fingerprint P [43] into some output $\pi_i(P)$. For each plaintext chunk M, FeatureSpy computes Rabin fingerprints over 64-byte sliding windows of chunk data (each sliding window returns one Rabin fingerprint). We compute $\pi_i(P)$ for each Rabin fingerprint P as follows:

$$\pi_i(P) = a_i * P + m_i \mod 2^{64}, \text{ for } 1 \le i \le N. \tag{1}$$

FeatureSpy derives N *sub-features*, in which the i-th sub-feature is the Rabin fingerprint P if $\pi_i(P)$ is the maximum over the Rabin fingerprints across all sliding windows. It computes a *feature* by concatenating multiple (e.g., four [47]) sub-features, and represents each plaintext chunk by $S = \frac{N}{4}$ features. Let $N = 12$ [47] (i.e., $S = 3$), P_i be the i-th sub-feature ($1 \le i \le 12$) and F_j be the j-th feature ($j = 1,2,3$). We have:

$$F_j = \mathbf{H}(P_{4j-3}||P_{4j-2}||P_{4j-1}||P_{4j}), \text{ for } j = 1,2,3, \tag{2}$$

for some cryptographic hash function $\mathbf{H}(\cdot)$. Two chunks with more common features are more likely to be similar.

Key generation. FeatureSpy generates a *feature key* based on the S features, so that similar chunks are likely to share the same feature key. This is a critical requirement to preserve similarity in encryption (§IV-B). Specifically, FeatureSpy concatenates all S features, and computes the feature key based on the cryptographic hash of the concatenation. Note that two chunks have an identical feature key only if their content differences do not alter *any* of the S features. Thus, a small S leads to a loose key generation criterion and tends to generate the same feature key for a large number of chunks. On the other hand, with a large S, FeatureSpy generates the same feature key only for the highly similar chunks with very few changed contents. In §VI-B, we will study how the choice of S impacts the trade-off between the detection accuracy and false positive rate in attack detection.

B. Similarity-preserving Encryption

To allow attack detection based on ciphertext chunks (§IV-C), FeatureSpy needs to preserve the similarity of the original plaintext chunks after encryption and detect if there exist many ciphertext chunks generated from similar plaintext chunks. Recall that MLE destroys content similarity, since the MLE key is generated based on the whole content of each plaintext chunk (§III-C). Our idea is to perform encryption based on the feature key. Since similar chunks are likely to have the same feature key (§IV-A), this can preserve content similarity.

One straightforward (but with security limitations) approach is *feature-based encryption (FBE)* [53], which directly encrypts each plaintext chunk with its feature key based on the cipher block chaining (CBC) block cipher mode. Specifically, similar chunks have few content changes and are likely to have identical blocks starting from the beginning of chunk data. Since the CBC encryption of each data block (16 bytes long) depends on the encryption of the previous block, if two ciphertext chunks have multiple initial blocks in common, we can deduce that the ciphertext chunks are likely to be generated from similar plaintext chunks. However, FBE is vulnerable to key compromise, since a feature key is the same for all chunks with an identical set of features. A malicious client can use a compromised feature key to fully decrypt many similar chunks, even though some of the chunks are unauthorized to access.

We propose *similarity-preserving encryption (SPE)*, which extends MLE with similarity preservation. Our idea is to encrypt only a small part of chunk content with feature key to preserve similarity, while the remaining large part is still protected by the MLE key to defeat against key compromise (note that the compromise of the MLE key of one chunk does not leak the information about any other chunk [18]). Specifically, due to the small content differences of similar chunks, SPE samples the first 32 bytes (with 0.4% of an 8 KiB chunk) from each plaintext chunk called the *indicator*. It encrypts the indicator of each plaintext chunk with the feature key, and encrypts the remaining large part (with 99.6% of an 8 KiB chunk) of chunk content with the MLE key. Similar chunks are likely to have the same indicator, and their encrypted indicators are also the same due to the same feature key. We can find the similar chunks by verifying if their encrypted indicators (i.e., the first 32 bytes of each ciphertext chunk) are identical. Note that SPE still achieves storage savings with deduplication, since identical chunks also have the same set of features, and hence the same feature key and MLE key.

SPE defends against key compromise by mitigating the information leakage as follows. Our insight is that if the feature key of a compromised chunk (say, M) is leaked, it can only be used to decrypt the indicators of the similar chunks that have the same set of features as M. Such similar chunks are likely to have the same indicator as M; in this case, SPE does not allow an adversary to learn a different indicator and it does not incur additional information leakage beyond the indicator. In addition, the remaining large part of M is still encrypted by the MLE key and protected under MLE [19].

C. Shielded Attack Detection

FeatureSpy proposes *shielded attack detection* to prevent a malicious client from tampering with in-memory operations and bypassing the detection phase (§III-B). Our idea is to couple attack detection and PoW together in the enclave, such that valid PoW proofs are truthfully generated for the authenticated chunks that have been examined for attack detection. Since the PoW proofs need to be verified before deduplication (§II-A), the malicious client cannot skip the detection phase to process forged contents. We present the design details as follows.

Attack detection. A straightforward design of FeatureSpy is to count the indicator of each ciphertext chunk in the enclave and report an attack if there exist many ciphertext chunks with the same indicator. However, it is infeasible to process all ciphertext chunks of a backup snapshot in attack detection and report the attack until the snapshot is completely processed, as the attack has already made damages before being detected. Also, as a snapshot typically has a large size, counting the indicators of all ciphertext chunks increases the enclave size.

FeatureSpy opts to detect attacks on the ciphertext chunks on a *per-batch* basis. The idea is that if a tampered snapshot has many similar chunks, it must have at least one batch that contains a large fraction of similar chunks. Specifically, the enclave manages a hash table to track how many ciphertext chunks in a batch have the same indicators (the default batch size $W = 16\,\text{K}$). Each hash table entry maps a unique indicator (32 bytes long) to the number of times (4 bytes long) that the indicator occurs across different ciphertext chunks in the batch. Note that the hash table has a size up to $16\,\text{K} \times$ (32 bytes + 4 bytes) $\approx 0.6\,\text{MiB}$ and adds negligible EPC overhead.

For each ciphertext chunk, the enclave queries the hash table based on the chunk's indicator. If an indicator does not exist, it adds the indicator into the hash table and initializes the count as one; otherwise, it increments the count by one. If the count reaches a pre-defined threshold T (e.g., 1% by default) of the batch size W, the enclave reports an attack and aborts the PoW operation (see below) for the client. After processing a batch of ciphertext chunks, the enclave clears all hash table entries to process the next batch of ciphertext chunks.

PoW. FeatureSpy implements PoW inside the enclave [45]. Specifically, for each examined ciphertext chunk of a batch, the enclave computes its fingerprint, and generates a signature based on the concatenation of all chunk fingerprints of the batch. The cloud checks if each fingerprint corresponds to a stored ciphertext chunk (i.e., deduplication) only when the signature is successfully verified.

D. Security Analysis

Confidentiality against a compromised cloud. We have discussed the security improvement of SPE over FBE when keys have been compromised (§IV-B). We now focus on the case that the keys are not leaked beforehand. Bellare et al. [19] have shown that any polynomial-time adversary cannot distinguish the MLE-based ciphertext of a plaintext chunk from a random value when the plaintext chunk is drawn from a large space (i.e., plaintext chunks are *unpredictable*). SPE extends MLE by encrypting the indicator with the feature key, and preserves the security of MLE if any feature is unpredictable. Specifically, under SPE, it is infeasible to enumerate all possible feature keys by a polynomial-time adversary. Then the encrypted indicator is also indistinguishable from a random value if the underlying symmetric encryption is secure. Note that we can further relax the unpredictable assumption via server-aided key generation [18] (see §V for implementation details).

Robustness against a malicious client. We have discussed that an adversary cannot forge chunks to bypass the detection phase (§IV-C). We now focus on the robustness against other malicious actions (§III-B).

- **Case 1: Tampering with unprotected operations.** A malicious client may manipulate features, indicators, and keys, in order to cheat the enclave for passing detection. However, these manipulations are not helpful to learn contents from a benign client that follows our design, since they lead to distinct ciphertext chunks with those produced by regular SPE (applied by the benign client). This prohibits deduplication, and learning-content attacks that rely on the information leakage of source-based deduplication are impossible.
- **Case 2: Tampering with data processing.** A malicious client may carefully inject forged files, such that each batch just includes a small fraction of similar chunks. FeatureSpy mitigates learning-content attacks by slowing down the attack procedure. Suppose that an adversary needs to generate a number of similar chunks to enumerate all possible contents. Without FeatureSpy, it can fill each batch with W similar chunks for the attack (recall that W is the batch size; see §IV-C); with FeatureSpy, it can only submit up to $W \times T$ similar chunks in a batch without being detected (recall that T is the fractional threshold for attack detection; see §IV-C), meaning that the adversary needs to generate $1/T$ times more batches in order to generate the same number of similar chunks to enumerate all possible contents. A smaller T implies that the adversary needs a longer time to launch learning-content attacks, yet FeatureSpy may have a higher false positive rate in attack detection.

V. Implementation

We implement a FeatureSpy prototype based on SGXDedup [45], a state-of-the-art SGX-based secure deduplicated storage system. SGXDedup builds on server-aided MLE [18], which maintains a *key manager* to manage a global secret. The key manager generates the MLE key for each plaintext chunk based on both the chunk fingerprint and the global secret to defend against offline brute-force attacks [18]. SGXDedup also deploys a client-side enclave and implements source-based deduplication with PoW inside the enclave (§IV-C) to defend against side-channel attacks. However, SGXDedup does not address learning-content attacks.

FeatureSpy augments SGXDedup with the detection of learning-content attacks. To address the unpredictability assumption for both chunks and features (§IV-D), the client performs server-aided key generation for MLE keys (like SGXDedup) and feature keys. To boost performance, the client parallelizes feature extraction with multiple (e.g., three by default) threads and pipelines the processes of chunking, key generation, encryption, attack detection, and uploads. Also, the cloud manages non-duplicate ciphertext chunks in units of *containers* with $8\,\text{MiB}$ each for storage, and maintains an in-memory least-recently-used cache ($1\,\text{GiB}$) to hold the most recently restored containers. For each download request, it first searches for the containers in the cache, and retrieves them from disk only if they are not in the cache.

VI. Evaluation

A. Datasets

Synthetic datasets. We consider a set of synthetic snapshots to study the trade-off between the detection accuracy and false

Datasets	Snapshots	Raw Size	Deduplication Ratio
Linux	84	42.7 GiB	3.8
GCC	85	33.0 GiB	7.2
CouchDB	83	22.9 GiB	1.8
Gitlab	79	74.4 GiB	1.1
FSL	10	407.5 GiB	9.3
MS	10	902.6 GiB	5.6

TABLE II: Characteristics of real-world datasets. The deduplication ratio is defined as the ratio between the size of pre-deduplicated data and the size of post-deduplicated data; a higher deduplication ratio means that the dataset has more content redundancies.

positive rate of attack detection (Exp#1). Suppose that each chunk has a size of 8 KiB. We create a 1 GiB *base snapshot* with fully random chunks (i.e., no duplicate or similar chunks). We use the base snapshot to generate two types of synthetic snapshots. First, we randomly inject a configurable number of forged files into the base snapshot, so as to simulate the case that an adversary mixes forged files and legitimate contents in uploads. Here, we fix each forged file with only a single chunk, and control the similarity of different forged files by the number of changed positions in the chunk and the number of different bytes in each changed position. We do not consider larger forged files, since the adversary needs to enumerate additional similar chunks and is more likely to be caught by FeatureSpy. Second, we randomly replace a fraction of chunks in the base snapshot by similar chunks, so as to simulate real-world non-compromised snapshots. We use such snapshots to study the false positive rates in different FeatureSpy instances.

Real-world datasets. We consider six real-world datasets (Table II): (i) *Linux* [7], which includes 84 snapshots from the stable versions (between v2.6.11 and v5.13) of Linux source code; (ii) *GCC* [3], which includes 85 snapshots from the release versions (between v4.0.0 and v12.1.0) of GNU GCC source code; (iii) *CouchDB* [2], which includes 83 docker images of CouchDB in the versions between v2.5.2 and v6.6.2; (iv) *Gitlab* [4], which includes 79 docker images of Gitlab-ce in the official release versions between v14.0.0-ce.0 and v14.9.4-ce.0; (v) *FSL* [9], which includes 10 weekly home directory backup snapshots from a shared network file system; and (vi) *MS* [36], which includes 10 windows file system snapshots with a logical size of about 100 GiB each. We will use Linux, GCC, CouchDB, and Gitlab snapshots to evaluate the detection accuracy of FeatureSpy when the malicious client mixes the forged employment letters of our case study (§II-B) with real-world uploads (Exp#2). Also, FSL and MS are large-scale workloads but only include chunk metadata rather than actual data, so we reproduce the chunks for both traces to evaluate the performance of FeatureSpy (Exp#4).

B. Detection Analysis

Default configuration. We configure FeatureSpy as follows. For each plaintext chunk, we extract S = 3 features to perform feature key generation (except Exp#1 that varies S to study the trade-off of different FeatureSpy instances). To perform attack detection, we fix the size of the indicator as 32 bytes and W = 16 K.

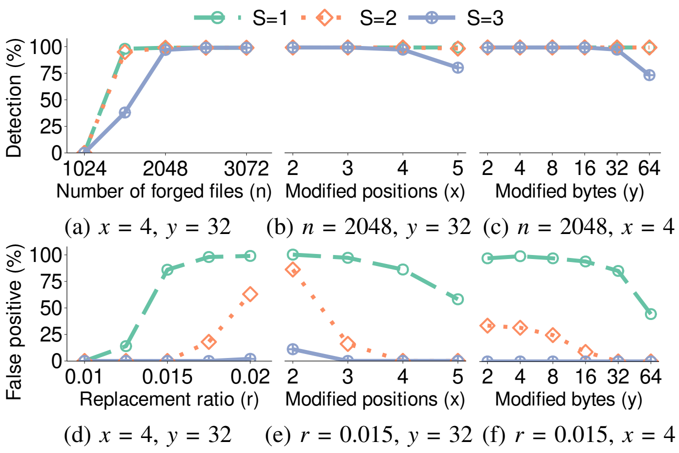

Fig. 4: (Exp#1) Trade-off study between detection accuracy and false positives of different FeatureSpy instances.

Exp#1 (Trade-off study). We evaluate the trade-off of different FeatureSpy instances configured by S. We first focus on the synthetic snapshots (§VI-A) that are mixed with a varying number n of 8-KiB forged files, which are configured with x differed positions and y different bytes in each differed position. For fair comparison, we fix the fractional threshold T of all instances as 1%, and measure the detection accuracy by the *detection rate*, defined as the ratio between the number of such mixed snapshots that are successfully detected by FeatureSpy and the total number of mixed snapshots. Specifically, we randomly generate 100 mixed snapshots and evaluate the detection rates of FeatureSpy instances (S = 1, 2, and 3).

Figure 4(a) shows the results when we fix x = 4 and y = 32, and vary n from 1,024 to 3,072. The detection rates of all instances increase with the number of forged files, since a large n injects more similar chunks. Also, the instances S = 1 and 2 can effectively report the existence of attacks (e.g., the detection rate is above 95%) even the malicious client only injects a small number (e.g., 1,536) of forged files, since a small S tends to assign identical feature keys to many chunks (§IV-A). This increases the probability of detecting similar chunks by FeatureSpy. Figures 4(b) and 4(c) show the detection rate when we fix n = 2,048 and vary x and y, respectively. A large x or y implies small similarity among the forged files. Thus, the detection rates of all FeatureSpy instances gradually decrease, since a small number of similar chunks can be found. Compared to the other two instances, the detection rate of S = 3 decreases more significantly. On the other hand, even the forged files have $\frac{64 \text{ bytes} \times 4 \text{ positions}}{8 \text{ KiB}} = 3.1\%$ different contents (Figure 4(c)), the instance S = 3 can detect attacks with at least the probability of 73%.

We further let FeatureSpy process the other type of synthetic snapshots (§VI-A), each of which replaces the chunks in the base snapshot by a fraction r of (similar) chunks. Similarly, we configure the replacement chunks with x differed positions in chunk contents and y different bytes in each differed position, in order to characterize different similarities. We randomly generate 100 such snapshots, and measure the *false positive*

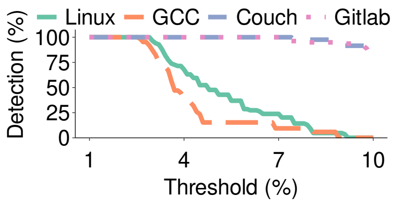

Fig. 5: (Exp#2) Case study of attack detection. We show the detection rate of FeatureSpy, and its default configuration does not introduce any false positives in the case.

Procedure/Step		FeatureSpy	SGXDedup
Chunking		2.12 ± 0.006	2.12 ± 0.006
Feature extraction		9.85 ± 0.02	-
Fingerprinting		1.81 ± 0.002	1.81 ± 0.002
Key generation		0.73 ± 0.02 (0.49 ± 0.01)	0.73 ± 0.02 (0.49 ± 0.01)
Encryption		1.22 ± 0.001	1.22 ± 0.001
In Enclave	Detection	0.04 ± 0.005	-
	PoW	1.86 ± 0.004	1.86 ± 0.004
Deduplication		0.55 ± 0.02	0.55 ± 0.02
Transfer		1.16 ± 0.03 (0.04 ± 0.001)	1.16 ± 0.03 (0.04 ± 0.001)

TABLE III: (Exp#3) Time breakdown per 1 MiB of synthetic file data processed in a single thread (unit: ms). We average the results over 10 runs and include the 95% confidence intervals from *Student's t-Distribution*. Except explicitly specified in parentheses, the consumed time in the second upload is identical with that in the first upload.

rate, defined as the ratio between the number of snapshots that FeatureSpy misjudges and the total number (i.e., 100) of snapshots.

Figure 4(d) presents the results, when we fix $x = 4$ and $y = 32$, and vary r from 0.01 to 0.02 (informed by the characteristics of real-world snapshots; see Figure 2). The false positive rate increases, since a large r implies more legitimate similar chunks in snapshots. However, the $S = 3$ instance keeps a low false positive rate (below 2%), since it only detects highly similar chunks. Similarly, in Figure 4(e) (where $r = 0.015$ and $y = 32$, and x varies) and Figure 4(f) (where $r = 0.015$ and $x = 4$, and y varies), when we reduce the similarity among the replacement chunks (by increasing x or y), the false positive rate decreases. To sum up, compared with the other instances, the $S = 3$ instance effectively balances the detection rate (at least 73% even 3.1% contents are modified among the forged chunks) and false positive rate (up to 11%).

Exp#2 (Case study of attack detection). We extend the case study in §II-B to study how FeatureSpy detects the learning-content attacks of inferring salaries and sign-on bonuses. Recall that the adversary forges $101 \times 31 = 3131$ employment letters, where the annual salary and sign-on bonus have 101 and 31 possible values, respectively (§II-B). To make detection more challenging, we evenly insert the forged letters into each real-world snapshot, perform chunking on each individual file of the mixed snapshot (that mixes both forged employment letters and legitimate chunks), and further apply SPE on the chunks. Informed by Exp#1, we only focus on $S = 3$ here.

Figure 5 shows the detection rate for the mixed snapshots, when we vary the threshold T from 1% to 10%. The detection rate generally decreases, since FeatureSpy needs to find more similar chunks to report the existence of attacks. Even so, our default configuration of $T = 1\%$ can detect all mixed snapshots. We also let FeatureSpy process each *raw snapshot* without forged letters, and evaluate the false positive rate. We find that FeatureSpy does not have any false positives even though we configure a small T (e.g., 1%). The reason is that the similar chunks in raw snapshots are too few to trigger the detection of the attack.

C. Performance Evaluation

Setup. We deploy FeatureSpy in a LAN testbed that includes multiple machines to run the cloud, key manager, and clients. Each machine is equipped with an eight-core 2.9 GHz Intel Core i7-10700 CPU, a 4 TB 7200 RPM Seagate Exos SATA HDD and 32 GB RAM. All machines are connected via a 10 Gbps switch and run Ubuntu 20.04.3. In addition to the default configuration (§VI-B), we configure three threads in FeatureSpy prototype to extract the features of plaintext chunks in parallel (except Exp#3 in which we first conduct microbenchmarks on the performance of each step in a single thread), and a 1 GiB container cache to improve the download performance (§V). Our goal is to show that FeatureSpy incurs small performance overhead over SGXDedup [45], which cannot defend against learning-content attacks (§V).

Exp#3 (Microbenchmarks). We conduct microbenchmarks by deploying a client, a key manager, and a cloud in distinct machines. We generate a 2 GiB file of synthetic data with random contents (i.e., no duplicate or similar chunks), and load the file into the client's memory before each test. We configure the client with a single thread to upload the same 2 GiB file twice, and further download the file. We evaluate the processing time of different upload steps, including: (i) *chunking*, which partitions the input file into variable-size plaintext chunks; (ii) *feature extraction*, which extracts the features of each plaintext chunk; (iii) *fingerprinting*, which computes the fingerprint of each plaintext chunk; (iv) *key generation*, which generates both feature keys and MLE keys; (v) *encryption*, which encrypts each plaintext chunk; (vi) *detection*, which detects learning-content attacks; (vii) *PoW*, which proves the ownership of each ciphertext chunk; (viii) *deduplication*, in which the cloud detects duplicate chunks; (ix) *transfer*, which transmits non-duplicate ciphertext chunks and the file recipe.

Table III compares the processing times (per 1 MiB file data) of FeatureSpy and SGXDedup [45]. The detection step is efficient, and takes up to 0.2% of the overall time in uploads. The feature extraction step is expensive due to the computational overhead of the similarity detection scheme [47]; it takes 50.9% and 54.8% of the overall time in the first and second uploads, respectively. However, we can accelerate feature extraction via multi-threading.

Figures 6(a) and 6(b) further present the speeds of the first and second uploads, respectively, by varying the number of threads to extract features (§V). SGXDedup keeps at 297.1 MiB/s in the first upload and 304.3 MiB/s in the second upload, since it does not extract features. The first

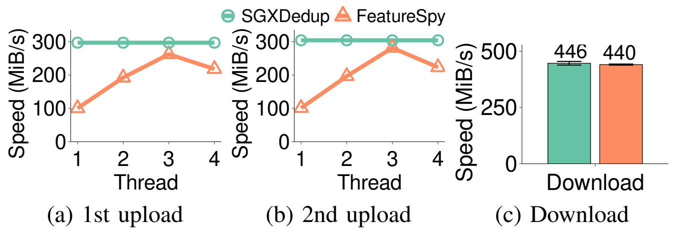

Fig. 6: (Exp#3) Microbenchmarks of multi-threading in upload and download.

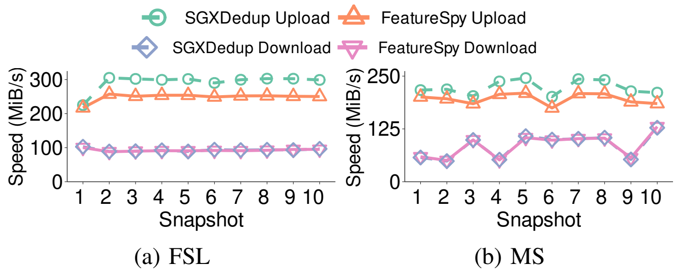

Fig. 7: (Exp#4) Trace-driven performance.

(second) upload speed of FeatureSpy increases to 262.6 MiB/s (281.6 MiB/s) with three threads due to parallel feature extraction, and decreases to 218.1 MiB/s (223.4 MiB/s) with four threads due to resource contention among threads. By exploiting multi-threading, it incurs a speed drop over SGXDedup with 11.6% in the first upload and 7.5% in the second upload. Figure 6(c) compares the download speed. FeatureSpy incurs 1.3% speed drop, since it decrypts each chunk with both the MLE key and feature key.

Exp#4 (Trace-driven performance). We compare the performance of FeatureSpy and SGXDedup using the real-world FSL and MS snapshots. Since our snapshots only contain chunk fingerprints and sizes without chunk content (§VI-A), we reconstruct each plaintext chunk by repeatedly writing its fingerprint into a spare chunk with the specified size. We first upload the snapshots one by one, and then download them in the same order of upload. Note that the original SGXDedup [45] does not have the container cache (§V). For fair comparison, we implement an in-memory cache for SGXDedup to buffer the most recently restored containers, and configure the cache with the same size (1 GiB) as in FeatureSpy.

Figure 7 presents the results. After the first FSL snapshot (224.8 MiB/s for SGXDedup and 216.9 MiB/s for FeatureSpy), both SGXDedup and FeatureSpy achieve high speeds (at least 298.9 MiB/s for SGXDedup and 250.1 MiB/s for FeatureSpy), since they do not need to transfer the cross-snapshot redundancies that take a large fraction in FSL. The download speed is generally steady (88.7-102.6 MiB/s for SGXDedup and 88.0-100.2 MiB/s for FeatureSpy). On average, compared to SGXDedup, FeatureSpy decreases the upload speed by 15.0% and the download speed by 0.8%.

Compared to FSL, the upload speeds of both systems in MS drop by about 21.0%, since MS contains many unique chunks and generates a large fingerprint index (implemented via LevelDB [6] in both SGXDedup and FeatureSpy). This aggravates the overhead of index queries. Also, the download speeds of both systems in MS fluctuate across snapshots, since some snapshots have more non-duplicate chunks and may be stored in the consecutive regions (i.e., less fragmented [35]) that can be quickly accessed via sequential reads.

VII. Related Work

Secure deduplication approaches. In §II-A, we review the basic primitives for secure deduplication. MLE preserves data confidentiality and is extensively studied from the security [10], [17], [31] and system [12], [18], [32], [33], [42], [45], [46] perspectives. FeatureSpy proposes SPE to augment MLE with similarity preservation, so as to support similarity detection on ciphertext chunks. PoW prevents a malicious client from compromising data ownerships, but it cannot address learning-content attacks (§II-B). FeatureSpy complements PoW by proactively detecting attacks. Previous studies [26], [33] prevent learning-content attacks by allowing to transfer duplicate chunks. FeatureSpy performs pure source-based deduplication without transferring duplicate chunks.

SGX-based secure storage. SGX has been widely used to strengthen the security of storage systems, such as file systems [11], [48], outsourced databases [24], [41], [49], and key-value stores [15], [16], [30], [38]. This paper focuses on secure deduplicated storage. S2Dedup [37] and DEBE [54] manage a cloud-side enclave to perform target-based deduplication, while FeatureSpy focuses on addressing a malicious client in source-based deduplication. SGXDedup [45] leverages SGX to improve the performance of PoW. FeatureSpy extends SGXDedup with the capability of detecting learning-content attacks.

VIII. Conclusion

This paper presents FeatureSpy, a secure deduplicated storage system that can effectively detect learning-content attacks. It builds on the insight that a malicious client generates many similar chunks for attacks, and proposes SPE to support similarity detection on ciphertext chunks, as well as shielded attack detection to prevent the malicious client from evading detection. FeatureSpy not only effectively detects learning-content attacks, but also incurs small performance overhead compared to SGXDedup [45].

Acknowledgements

This work was supported in part by the National Key R&D Program of China (2022YFB3103500), National Natural Science Foundation of China (61972073, 62002049, 61872057, U19A2066), Natural Science Foundation of Sichuan (2023NSFSC0472), Key Research Funds of Sichuan Province (2021YFG0167, 2020YFG0298), Sichuan Science and Technology Program (2020JDTD0007, 2022YFG0037), Fundamental Research Funds for Chinese Central Universities (ZYGX2020ZB027, ZYGX2021J018, ZYGX2020J017), Innovation and Technology Fund (GHX/076/20), and Research Matching Grant Scheme.

References

[1] Alibaba cloud. https://www.alibabacloud.com/.

[2] CouchDB. https://couchdb.apache.org.

[3] GCC. https://github.com/gcc-mirror/gcc.

[4] Gitlab docker images. https://hub.docker.com/r/gitlab/gitlab-ce.

[5] Intel(R) software guard extensions. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html.

[6] LevelDB. https://github.com/google/leveldb.

[7] The linux kernel archives. https://www.kernel.org/.

[8] Offer letter between Google and Patrick Pichette dated June 6, 2008. https://www.sec.gov/Archives/edgar/data/1288776/000119312508140342/dex101.htm.

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

[10] M. Abadi, D. Boneh, I. Mironov, A. Raghunathan, and G. Segev. Message-locked encryption for lock-dependent messages. In *Proc. of CRYPTO*, 2013.

[11] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. OBLIVIATE: A data oblivious filesystem for Intel SGX. In *Proc. of NDSS*, 2018.

[12] F. Armknecht, J.-M. Bohli, G. O. Karame, and F. Youssef. Transparent data deduplication in the cloud. In *Proc. of ACM CCS*, 2015.

[13] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell, et al. SCONE: Secure linux containers with Intel SGX. In *Proc. of USENIX OSDI*, 2016.

[14] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proc. of ACM CCS*, 2007.

[15] M. Bailleu, D. Giantsidi, V. Gavrielatos, D. L. Quoc, V. Nagarajan, and P. Bhatotia. Avocado: A secure in-memory distributed storage system. In *Proc. of USENIX ATC*, 2021.

[16] M. Bailleu, J Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. SPEICHER: Securing LSM-based key-value stores using shielded execution. In *Proc. of USENIX FAST*, 2019.

[17] M. Bellare and S. Keelveedhi. Interactive message-locked encryption and secure deduplication. In *Proc. of PKC*, 2015.

[18] M. Bellare, S. Keelveedhi, and T. Ristenpart. DupLESS: Server-aided encryption for deduplicated storage. In *Proc. of USENIX Security*, 2013.

[19] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. In *Proc. of EuroCrypto*, 2013.

[20] J. Black. Compare-by-hash: A reasoned analysis. In *Proc. of USENIX FAST*, 2006.

[21] A. Broder. On the resemblance and containment of documents. In *Proc. of Compression and Complexity of Sequences*, 1997.

[22] T. Dinh Ngoc, B. Bui, S. Bitchebe, A. Tchana, V. Schiavoni, P. Felber, and D. Hagimont. Everything you should know about Intel SGX performance on virtualized systems. In *Proc. of ACM SIGMETRICS*, 2019.

[23] J. R. Douceur, A. Adya, W. J. Bolosky, P. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proc. of IEEE ICDCS*, 2002.

[24] S. Eskandarian and M. Zaharia. ObliDB: Oblivious query processing for secure databases. In *Proc. of VLDB*, 2017.

[25] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg. Proofs of ownership in remote storage systems. In *Proc. of ACM CCS*, 2011.

[26] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, 2010.

[27] D. Harnik, E. Tsfadia, D. Chen, and R. Kat. Securing the storage data path with SGX enclaves. https://arxiv.org/abs/1806.10883, 2018.

[28] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proc. of ACM SYSTOR*, 2009.

[29] A. Juels and B. S. Kaliski, Jr. PORs: Proofs of retrievability for large files. In *Proc. of ACM CCS*, 2007.

[30] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh. ShieldStore: Shielded in-memory key-value storage with SGX. In *Proc. of ACM EuroSys*, 2019.

[31] J. Li, G. Wei, J. Liang, Y. Ren, P. P. C. Lee, and X. Zhang. Revisiting frequency analysis against encrypted deduplication via statistical distribution. In *Proc. of IEEE INFOCOM*, 2022.

[32] J. Li, Z. Yang, Y. Ren, P. Lee, and X. Zhang. Balancing storage efficiency and data confidentiality with tunable encrypted deduplication. In *Proc. of ACM Eurosys*, 2020.

[33] M. Li, C. Qin, and P. Lee. CDStore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal. In *Proc. of USENIX ATC*, 2015.

[34] D. Lie. Minimizing the TCB. In *Proc. of USENIX Security*, July 2005.

[35] M. Lillibridge, K. Eshghi, and D. Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proc. of USENIX FAST*, 2013.

[36] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *Proc. of USENIX FAST*, 2011.

[37] M. Miranda, T. Esteves, B. Portela, and J. Paulo. S2Dedup: SGX-enabled secure deduplication. In *Proc. of ACM SYSTOR*, 2021.

[38] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *Proc. of IEEE S&P*, 2018.

[39] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *Proc. of USENIX Security*, 2011.

[40] S. Pinto and N. Santos. Demystifying ARM TrustZone: A comprehensive survey. *ACM Computing Surveys*, 51(6):130:1–130:36, 2019.

[41] C. Priebe, K. Vaswani, and M. Costa. Enclavedb: A secure database using sgx. In *Proc. of IEEE S&P*, 2018.

[42] C. Qin, J. Li, and P. Lee. The design and implementation of a rekeying-aware encrypted deduplication storage system. *ACM Transactions on Storage*, 13(1):9:1–9:30, 2017.

[43] M. C. Rabin. Fingerprint by random polynomials. Technical report, Center for Research in Computing Technology, Harvard University, 1981.

[44] A. Rao. Rising to the challenge—data security with intel confidential computing. https://community.intel.com/t5/Blogs/Products-and-Solutions/Security/Rising-to-the-Challenge-Data-Security-with-Intel-Confidential/post/1353141, 2022.

[45] Y. Ren, J. Li, Z. Yang, P. Lee, and X. Zhang. Accelerating encrypted deduplication via SGX. In *Proc. of USENIX ATC*, 2021.

[46] P. Shah and W. So. Lamassu: Storage-efficient host-side encryption. In *Proc. of USENIX ATC*, 2015.

[47] P. Shilane, M. Huang, G. Wallace, and W. Hsu. WAN optimized replication of backup datasets using stream-informed delta compression. In *Proc. of USENIX FAST*, 2012.

[48] S. Shinde, S. Wang, P. Yuan, A. Hobor, A. Roychoudhury, and P. Saxena. BesFS: A POSIX filesystem for enclaves with a mechanized safety proof. In *Proc. of USENIX Security*, 2020.

[49] Y. Sun, S. Wang, H. Li, and F. Li. Building enclave-native storage engines for practical encrypted databases. In *Proc. of VLDB*, 2021.

[50] Z. Sun, G. Kuenning, S. Mandal, P. Shilane, V. Tarasov, N. Xiao, et al. A long-term user-centric analysis of deduplication patterns. In *Proc. of IEEE MSST*, 2016.

[51] B. Toulas. New intel chips won't play blu-ray disks due to SGX deprecation. https://www.bleepingcomputer.com/news/security/new-intel-chips-wont-play-blu-ray-disks-due-to-sgx-deprecation/.

[52] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *Proc. of USENIX FAST*, 2012.

[53] S. Wu, Z. Tu, Z. Wang, Z. Shen, and B. Mao. When delta sync meets message-locked encryption: A feature-based delta sync scheme for encrypted cloud storage. In *Proc. of IEEE ICDCS*, 2021.

[54] Z. Yang, J. Li, and P. Lee. Secure and lightweight deduplicated storage via shielded deduplication-before-encryption. In *Proc. of USENIX ATC*, 2022.

[55] Y. Zhang, W. Xia, D. Feng, H. Jiang, Y. Hua, and Q. Wang. Finesse: Fine-grained feature locality based fast resemblance detection for post-deduplication delta compression. In *Proc. of USENIX FAST*, 2019.

[56] N. Zhao, H. Albahar, S. Abraham, K. Chen, V. Tarasov, D. Skourtis, L. Rupprecht, A. Anwar, and A. R. Butt. DupHunter: Flexible high-performance deduplication for docker registries. In *Proc. of USENIX ATC*, 2020.

[57] P. Zuo, Y. Hua, C. Wang, W. Xia, S. Cao, Y. Zhou, and Y. Sun. Mitigating traffic-based side channel attacks in bandwidth-efficient cloud storage. In *Proc. of IEEE IPDPS*, 2018.