Exploiting Multiple Similarity Spaces for Deduplication of Encrypted Container Images

TONG SUN, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
BOWEN JIANG, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
BORUI LI, Southeast University, China
JIAMEI LV, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
YI GAO, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
WEI DONG, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China

The growing popularity of encrypted container images in registries poses unique challenges for storage management due to the necessity for deduplication amidst rising image volumes. Traditional deduplication struggles with encrypted content, which inherently disguises duplicate data as distinct due to its randomized nature. Current advanced methods tackle this issue by decompressing images and applying message-locked encryption (MLE). However, these techniques face considerable challenges. Minor content changes can impair deduplication effectiveness, and decompressing layers increases storage requirements. Furthermore, this process negatively impacts both the speed at which users access the images and the overall system throughput.

We propose SimEnc, a high-performance and secure deduplication system for encrypted container images by exploiting multiple similarity spaces. SimEnc pioneers the integration of semantic hashing with MLE to effectively parse semantic relationships across layers, thereby increasing deduplication efficacy. This system incorporates a rapid selection mechanism for similarity spaces, offering enhanced flexibility over previous models that relied on full decompression. By adopting Huffman decoding to navigate new similarity spaces, SimEnc not only improves deduplication ratios but also enhances overall performance. Our experimental results demonstrate that SimEnc substantially reduces storage needs by up to 261.7% compared to encrypted serverless platforms and by 54.2% against plaintext registries, while also delivering superior pull latency metrics.

CCS Concepts: • **Security and privacy** → **Management and querying of encrypted data**; *File system security*; • **Information systems** → **Deduplication**; *Cloud based storage*; • **Networks** → *Cloud computing*; • **Software and its engineering** → *File systems management*.

Additional Key Words and Phrases: Docker registry, Container image

ACM Reference Format:
Tong Sun, Bowen Jiang, Borui Li, Jiamei Lv, Yi Gao, and Wei Dong. 2025. Exploiting Multiple Similarity Spaces for Deduplication of Encrypted Container Images. *J. ACM* 37, 4, Article 111 (August 2025), 31 pages. https://doi.org/XXXXXXX.XXXXXXX

Authors' Contact Information: Tong Sun, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China, tongsun@zju.edu.cn; Bowen Jiang, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China, jiangbw@zju.edu.cn; Borui Li, Southeast University, China, librchn@gmail.com; Jiamei Lv, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China, lvjm@zju.edu.cn; Yi Gao, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China, gaoyi@zju.edu.cn; Wei Dong, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China, dongw@zju.edu.cn.

ACM 1557-735X/2025/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Privacy concerns have catalyzed the adoption of encrypted container images within container registries, as evidenced by their burgeoning popularity [17, 23, 33]. Such encryption is primarily aimed at ensuring that only specific, authorized parties can access the contents. Notable implementations of encryption technologies, such as the Advanced Encryption Standard (AES) [70] is widely deployed in platforms like IBM Cloud [40] and AWS Lambda [11]. These platforms store container images as compressed layers; each contains the application's executable and all necessary dependencies [37]. While encrypted images are visible to all users, they are not executable or accessible to those without proper authorization [23].

With the container ecosystem's growth, the number of images managed by registries like Docker Hub has surged. As of the autumn of 2020, Docker Hub reportedly stored several hundred million images, cumulatively taking up over seven petabytes of space [61, 75]. A study on Docker Hub's dataset indicated that approximately 97% of the files across various layers were redundant [91], underscoring a critical need for effective deduplication strategies to optimize storage utilization.

The challenge in deduplicating encrypted images lies in the fundamental clash between the goals of encryption—rendering data indistinguishable—and deduplication, which relies on identifying and eliminating duplicate data [76]. To address this, AWS Lambda has adopted a sophisticated approach utilizing message-locked encryption (MLE) technology [7, 16, 21, 29, 31, 52, 54, 69, 80, 85]. This process involves decompressing the container image, segmenting it into uniform blocks, and computing the SHA256 hash for each block to generate a unique identifier. These identifiers then serve as keys to encrypt the blocks using AES [70], ensuring that identical blocks yield identical encrypted forms. This method effectively enhances the deduplication ratio[1], defined as the original dataset size divided by the size after deduplication, relative to when all layers remain compressed [89].

Unfortunately, our measurements (cf. §3) uncover two limitations in current state-of-the-art approaches [16, 89]. These approaches decompress container images before applying MLE for deduplication.

Limitation I. *Even minor modifications to the image content can hinder MLE deduplication, as they change the SHA256 hash value of the generated key.* The state-of-the-art MLE technique [31, 52, 79, 80] employs locality-sensitive hashing (LSH) [15, 41, 87] to generate identical keys for similar chunks. LSH functions generate similar data signatures for data blocks with similar bit patterns, which is called data sketching [74]. This LSH-based MLE approach derives a chunk's key from its sketch and segments the chunks into smaller sub-chunks. Consequently, identical sub-chunks from similar chunks encrypted with the same sketch can be deduplicated, improving the deduplication ratio. However, a recent study [67] shows that the state-of-the-art LSH technique [87] produces high false negative rates that generate different sketches for similar data blocks. In our analysis (cf. §3.1), we observe that 49.2% of similar data pairs in our Docker dataset resulted in different sketches. Consequently, the high false-negative rate in LSH-based MLE hinders the generation of identical keys for similar blocks, undermining storage deduplication.

Limitation II. *Although decompression restores the similarity of file contents, it leads to an increase in storage consumption after deduplication.* We identify existing works [16, 89] for container image deduplication operating in the decompressed similarity space, which completely decompresses (i.e., LZ77 decoding and Huffman decoding [30]) layers in the image for deduplication. We also define the space where compressed bytes are located as compressed similarity space. We conduct encrypted deduplication using LSH-based MLE on the 264GiB container image of IBM datasets [39].

[1]We define the deduplication ratio = $\frac{\text{original data-set size}}{\text{data-set size after deduplication}}$, which is calculated against the case when all layers are compressed [89].

Table 1. Comparison of SimEnc with related work

Works	Flexibility	Security	Deduplication ratio	Latency
DupHunter [89]	Medium	Low	Medium (plaintext)	Low
AWS Lambda [16]	Low	High	Medium (ciphertext)	High
SimEnc (Ours)	High	High	High (ciphertext)	Low

The result shows that although it could deduplicate 357GiB of data after decompression, the system still required storage of 283GiB of duplicates. Furthermore, we note that duplicates cannot be compressed before encryption (for security reasons [18, 46, 85]) and after encryption as encrypted data are with high entropy [85]. Meanwhile, decompressing images before deduplication leads to two consequences: (i) as the view of clients, the image requires re-compression during restoration, increasing the client's pull latency [89]; (ii) as the view of service providers, in our measurements, it results in a 67% reduction in deduplication throughput compared to non-decompression. The state-of-the-art flexible container registry, DupHunter [89, 90], employs selective decompression of statistically popular layers to reduce client pull latency. However, this strategy compromises the deduplication ratio since the popular layers [37] would not be selected to decompress before deduplication.

In this paper, we propose **SimEnc**, a high-performance similarity-preserving encryption approach for deduplication of encrypted Docker images. We summarize our contributions as follows:

- We explore a new similarity space in Docker images by only using Huffman decoding, which we term as the *partially decoded space*. We first measure it as a new trade-off space of deduplication ratio and latency better than the existing completely decompressed space.
- We propose a fast similarity space selection mechanism that leverages the Huffman tree located at the header of each layer for similarity assessment. To balance the trade-off between deduplication ratio and throughput, we partially decode layers that are highly similar for block-level deduplication, whereas others undergo deduplication solely at the layer granularity.
- We propose a semantic-aware MLE technique, which is the first work to introduce semantic hashing in encrypted deduplication for improving the deduplication ratio. First, we exploit semantic-preserving learning to preserve the semantic information and utilize hashing contrastive learning to extract discriminative representations in partially decoded space. Second, we propose a similarity-preserving key generation mechanism to overcome the inability of semantic hashing to generate an identical sketch for similar chunks that could not be duplicated after encryption.
- We propose a privacy-preserving key generation mechanism that utilizes Trusted Execution Environments (TEEs) on cloud platforms to generate semantic keys for users. SimEnc diverges from previous models that required users to upload plaintext data. Instead, it keeps a cluster list within the TEE and conducts comparisons on semantic hashes to derive keys, ensuring that user privacy is not compromised.

We evaluate SimEnc on a 3-node cluster using real-world workloads and datasets. Table 1 illustrates the comparison of SimEnc with related work in terms of flexibility, security, deduplication ratio, and latency. In the highest deduplication mode, SimEnc outperforms both the state-of-the-art encrypted serverless platform (AWS Lambda [16]) and plaintext Docker registry (DupHunter [89]), reducing storage consumption by up to 261.7% and 54.2%, respectively. SimEnc also surpasses DupHunter in pull latency reduction (up to 27.7%) and can outperform AWS Lambda in end-to-end latency under low bandwidth conditions (below 50MB/s). In flexible mode, SimEnc further reduces storage consumption by 86.2% compared to DupHunter, with only a 7.3% increase in pull

latency overhead, which is practically unnoticeable to clients. Moreover, SimEnc is compatible with DupHunter's flexible mode and supports various other deduplication modes, offering diverse performance and storage savings trade-offs. Additionally, SimEnc can be seamlessly integrated into existing Docker registries and serverless platforms.

2 Background and Related Work

2.1 Encrypted Deduplication

Deduplication in plaintext is straightforward, but encryption, which randomizes content, complicates the process [76]. The message-locked encryption (MLE) [7, 16, 21, 29, 31, 52, 54, 69, 80, 85] is a cryptographic method designed to enable deduplication of encrypted data by generating encryption keys from the content of the messages themselves. A representative implementation of MLE is convergent encryption [7, 29], which uses the hash value (e.g., SHA256) of a message as the MLE key. AWS Lambda [16] deploys this MLE approach to deduplicate encrypted container images after decompression.

The state-of-the-art MLE technique is the locality sensitive hash (LSH)-based MLE [31, 52, 54, 80]. It employs LSH to generate chunk sketches, which we call super features [74]. LSH-based MLE computes the hash value $H_i(W_j)$ for each sliding window W_j, where j denotes the starting byte position of the window, and i is the feature number. The extracted features are calculated by the maximal hash value $Max(H_i(W_j))$. Then, it constructs super-features (SFs) by transposing n features [87]. Minor modifications in a chunk's contents can alter its SHA256 value, leading to a different key generation of the MLE. However, LSH-based MLE uses SFs of the chunk to extract chunk features, which tolerates these minor modifications [87], allowing for the generation of the same key. However, encrypting similar chunks using the same key leads to distinct encrypted chunks. To address this, existing works utilize the Content-Defined Chunking (CDC) [63, 82] technique to generate variable-length sub-chunks, and encrypt them with the same key. CDC employs a sliding window to compute a hash value (e.g., Rabin's fingerprint) of the data contained in the window. When the hash value satisfies the pre-defined condition, CDC determines the chunk boundaries, creating variable-size chunks based on the data.

2.2 Secure Deduplication with TEE

Recent innovations have integrated Trusted Execution Environments (TEEs) to bolster the security of deduplication systems. SGXDedup [69] employs Intel SGX to enhance the efficiency of encrypted deduplication through server-aided message-locked encryption (MLE), ensuring robust security. Conversely, DEBE [85] is the state-of-the-art approach that adopts a deduplication-before-encryption strategy. This method initially eliminates duplicates of commonly occurring data within the limited space of an SGX enclave, followed by the removal of any remaining duplicates externally. While these advancements underscore the potential of TEEs in securing deduplication frameworks, they also expose the difficulties of maintaining security alongside computational efficiency and scalability across varied environments. Distinct from these methodologies, SimEnc utilizes TEEs for the secure generation of semantic hash keys, thereby meticulously preserving user privacy.

2.3 Docker Registry

We first introduce the basics of building and distributing Docker images, then explore the state-of-the-art registries for both encrypted and plaintext images.

2.3.1 Image Building. In Docker, the process of building an image is initiated by the `docker build` command, which constructs the image according to a set of instructions specified in a Dockerfile [4]. Each instruction in the Dockerfile corresponds to a distinct operation that modifies the image, such

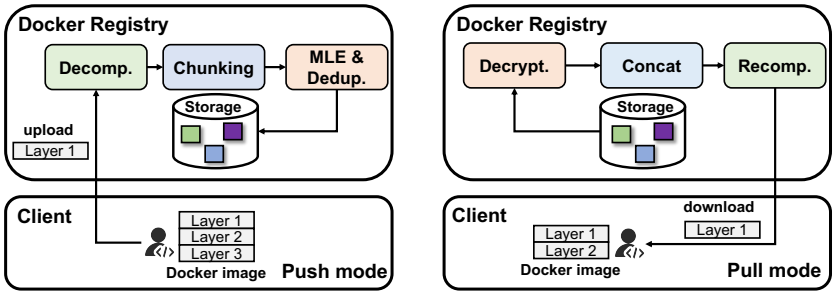


Fig. 1. Existing Docker registry for encrypted Docker image deduplication.

as installing packages or copying files. These instructions result in the creation of separate layers, where each layer encapsulates the changes introduced by a particular instruction. For example, a `RUN apt-get install` command generates a layer containing the installed files, while a `COPY` or `ADD` command creates layers that store the added files. Each layer represents a set of filesystem changes specified by a Dockerfile instruction. These layers are then stacked upon one another in the Docker image, allowing Docker to efficiently manage and reuse layers across different images.

After the layers are created, they are archived into tarballs using the tar utility, which consolidates the changes introduced in the layers. The resulting tarballs are then compressed with gzip, using the deflate algorithm. We will explain the deflate algorithm in detail in §2.5. The main goals of compression are to reduce the size of the image layers to save disk space and to speed up their transfer across networks.

2.3.2 Image Distribution. Once an image is built and compressed, it can be uploaded to a Docker registry with the `docker push` command. This process uploads the image layer by layer, storing each in the registry where it becomes accessible to multiple users or nodes. This layer-based distribution model enables efficient sharing of images across different systems. When a client (i.e., user or node) retrieves an image with the `docker pull` command, Docker checks if the required layers are already local. If they are, because they were previously downloaded or shared by other nodes, Docker skips re-downloading them, which saves time and network bandwidth.

Docker registries are primarily focused on storing and distributing container images. A registry provides a RESTful API [10] for Docker clients to push images to and pull images from the registry [26, 27]. Docker registries organize images into repositories, where each repository holds different versions or tags of an identical image, denoted as `repo-name:tag`. In these repositories, the registry maintains a manifest for each tagged image. This manifest is a JSON document detailing the container image's runtime settings (like environment variables) and the set of layers comprising the image. Each layer, a compressed archive file, is uniquely identified by a SHA256 digest calculated from its uncompressed form. When retrieving an image, the Docker client initially fetches the manifest, followed by the requisite layers not already on the client. Conversely, when uploading an image, the client uploads any new layers to the registry before the manifest.

Figure 1 shows a typical Docker registry [40] which contains encrypted images. In the client push mode, upon receiving a layer, the Docker registry decompresses it and divides it into fixed-size chunks. These chunks are then subject to encrypted deduplication using MLE. Conversely, in client pull mode, the encrypted chunks must first be decrypted and then concatenated with others to form an archived layer. Subsequently, this archived layer is re-compressed prior to being transferred to the client.

2.3.3 AWS Lambda. AWS Lambda [16] is the state-of-the-art registry for encrypted container image storage and distribution. Upon receiving a container image, it flattens each layer and splits it into fixed-size blocks, each of 512 KiB. These blocks are uploaded to the container registry and stored in a multi-tier distributed cache system. During function invocation, Lambda retrieves the required blocks from either the local cache or the origin storage (typically Amazon S3). If a block is not present in the cache, it is fetched from the origin and loaded into the local cache of the worker node. The caching system consists of both local caches and a shared availability zone level cache, which is employed to improve performance. Frequently accessed blocks are stored in memory for faster access, while less frequently used blocks are stored in flash memory. Lambda utilizes the MLE-based deduplication technique to ensure that identical content uploaded by different customers is not redundantly stored. This method derives an encryption key deterministically by hashing the content of each block, thereby ensuring confidentiality while optimizing storage efficiency. This deduplication strategy reduces storage overhead and minimizes data movement, all while maintaining strong data privacy guarantees.

2.3.4 DupHunter. DupHunter [89] is the state-of-the-art Docker registry designed to optimize both storage efficiency and layer retrieval performance for plaintext container images. The performance of registries is vital for clients, especially regarding the efficiency of layer retrieval (i.e., `pull` layer latency) [37, 89]. This aspect notably influences the time it takes to start a container [37]. DupHunter uses configurable deduplication modes to balance the trade-offs between storage space and retrieval latency. Specifically, it minimizes layer retrieval latency by leveraging a multi-tier storage hierarchy, where frequently accessed layers are kept in memory for faster access, while less frequently accessed layers are stored in secondary storage. Unlike AWS Lambda, which primarily relies on a fixed chunking mechanism for deduplication, DupHunter selectively decompresses layers based on their popularity before deduplication, allowing it to store popular layers more efficiently and reduce retrieval times. DupHunter also uses proactive caching strategies to predict which layers are likely to be requested and preemptively loads them into cache, further improving retrieval performance.

2.4 Encryped Container Image

Encrypted container images are typically associated with tools like containerd-imgcrypt [1], whose primary goal is to ensure confidentiality during image distribution. For example, containerd-imgcrypt extends Docker's or containerd's distribution by encrypting entire layers, preventing unauthorized parties from inspecting or tampering with their contents. Similarly, initiatives like Docker Content Trust (DCT) [3] and Cosign [2] provide authentication and integrity guarantees through cryptographic signatures, thus controlling who can push, pull, or verify images. However, these approaches treat encrypted layers as opaque data and do not perform intra-image or inter-image deduplication. While they ensure the confidentiality of the data, they do not address storage overhead reduction or distribution efficiency by leveraging redundancy across layers.

In contrast, SimEnc not only ensures the encryption of container images but also preserves the ability to detect redundancy and perform deduplication across images. This is achieved through similarity-preserving encryption, a key feature that distinguishes SimEnc from traditional encryption-only solutions. SimEnc's design specifically targets the problem of balancing high-performance similarity-based deduplication with encryption, which allows for optimized storage and efficient distribution while maintaining data confidentiality. While encryption ensures the confidentiality of the content, access control and authentication of who can pull, push, or verify the images are handled by existing mechanisms such as DCT [3].

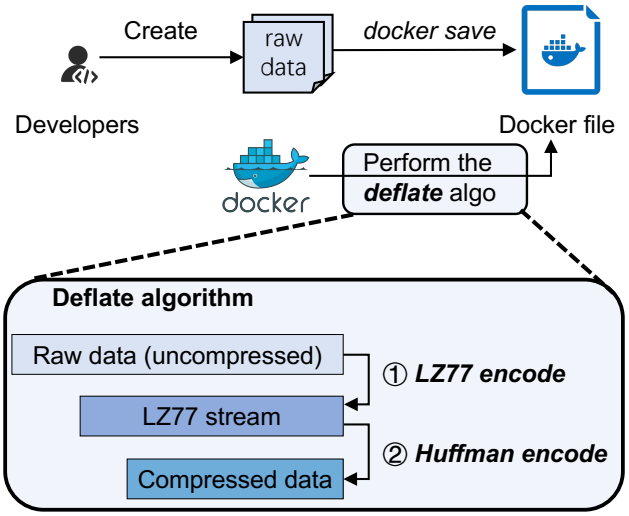


Fig. 2. The deflate process of Docker file.

2.5 Deflate Algorithm

Each layer of Docker images is archived using the `tar` and then compressed with the `gzip`[2] [30], which utilizes the deflate lossless compression algorithm. As shown in Figure 2, the deflate algorithm is a combination of LZ77 encoding and Huffman encoding [22, 32]. It operates in two primary stages, initially applying LZ77 to detect and encode repeating patterns within the data, followed by Huffman coding to optimally encode these patterns and literals based on their frequencies. The inflate algorithm [66] can flatten deflate streams by Huffman decoding and LZ77 decoding.

LZ77 encoding. LZ77 is a dictionary-based compression technique [92]. It reduces the data size by finding repeated sequences of strings and replacing them with references to previous occurrences of the same sequence. LZ77 utilizes tokens to represent sequences in the data. These tokens are categorized into two types: *Literal (LIT) Tokens*: These tokens directly encode single characters from the input data, denoted by a tuple where the first element (x) equals 0, the second element (y) equals 0, and the third element (z) is the character itself. *Reference (REF) Tokens*: These tokens identify repeated sequences by specifying a distance and length. The tuple for a REF token includes the distance back to the start of the repetition and the length of the repeated sequence. The distance and length are represented by x and y for the lower and upper 8 bits of the distance, respectively, and z for the length. These references consist of two parts: a distance (how far back from the current position) and a length of the repeated sequence.

Huffman encoding. In the subsequent stage, frequencies of the literals and sequences encoded by LZ77 are used to construct optimal Huffman trees [38]. This encoding produces a compact stream of data where frequently occurring sequences and literals use shorter codes. Each deflate stream has a compressed block (length and distance codes) which is a 286-dimension vector of Huffman tree [22].

[2]To the best of our knowledge, official Docker Hub images are compressed using `gzip`. While `zstd` compression is now available for Docker images, the key idea of SimEnc is not tied to any specific compression tool.

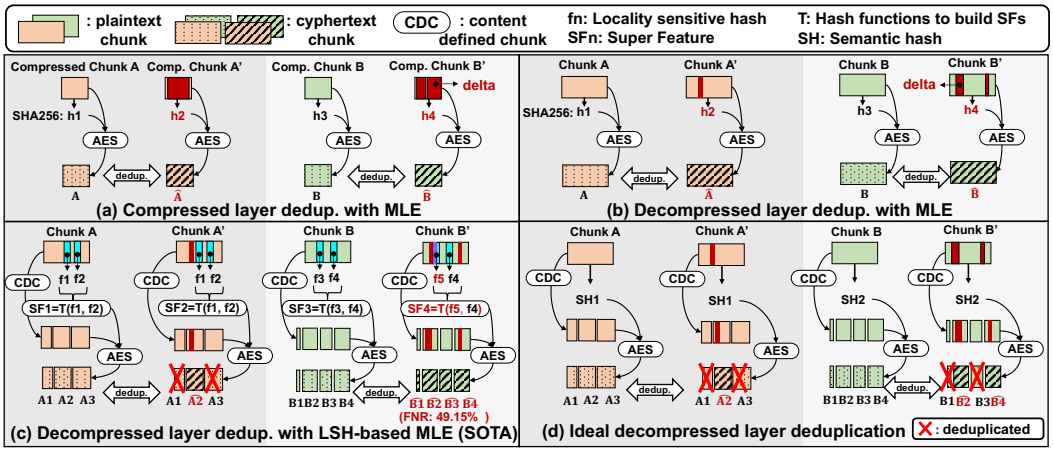


Fig. 3. The gap between existing message-locked encryption (MLE) works and the ideal.

3 Motivating Observations

The need and feasibility of SimEnc are based on two key observations: (i) existing MLE approaches tend to be highly sensitive to small changes in input (*high perturbation*), which results in a low deduplication ratio; (ii) a *partially decoded space* exists in Docker images where we can achieve a higher deduplication ratio and lower latency compared to the existing decompressed space.

3.1 Limitations of Existing Encrypted Deduplication Works

We now describe high-level ideas of MLE, the state-of-the-art LSH-based MLE, and the ideal encrypted deduplication. The deduplication approach is aligned with the AWS Lambda configuration, which divides layers into fixed-size chunks [16]. In Figure 3(a) and Figure 3(b), we make two observations: *(i)* it is difficult to obtain benefit from deduplicating two similar chunks in the compressed space because compression destroys the similarity [60]; *(ii)* the MLE employed in AWS Lambda [16] utilizes SHA256 hashes as keys. While decompression reveals more similarities, minor content changes hinder deduplication.

Figure 3(c) shows the LSH-based MLE performance in data deduplication. Chunk A produces features (f1 and f2) to calculate super feature (SF) and is divided into sub-chunks to address the shift boundary problem [80]. Chunk A' follows the same steps, but the feature is compromised by delta bytes. It creates a different SF from Chunk A, preventing deduplication. Although plaintexts of sub-chunks ($A1$ and $\hat{A1}$, $A3$ and $\hat{A3}$) are identical, the keys derived from SFs are distinct. To quantify such occurrences, we analyze 108,637 128KiB data blocks from real datasets (cf. §6). Compared to brute-force methods (e.g., using Xdelta [44] for chunk similarity calculations), we observe that 49.15% of chunk pairs showed over 50% byte-level similarity[3], yet their sketches significantly differed.

To the best of our knowledge, generating identical keys for similar chunks is difficult. The state-of-the-art approach to extract data features is semantic hash [50, 67, 78, 83], which can map infinite data into finite hash codes while preserving the semantic distance. It is widely used in image retrieval and recommendation systems. The ideal encrypted deduplication is shown in Figure 3(d). Ideally, only the semantic hash codes of similar chunks are identical, all identical sub-chunks could

[3]We define the byte-level similarity as $\frac{\text{delta size after delta compression}}{\text{original size}}$.

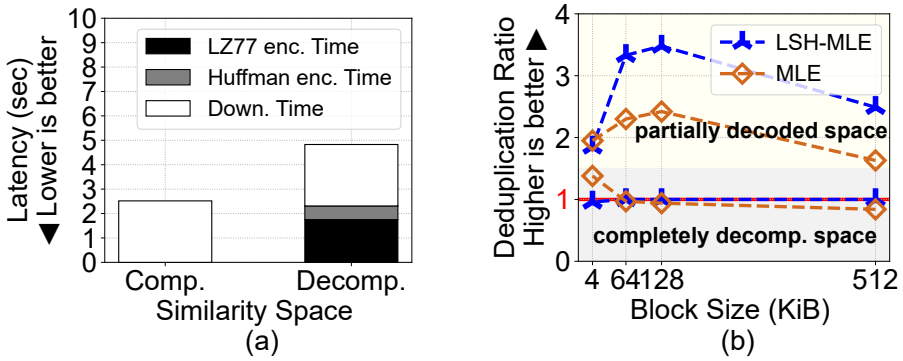


Fig. 4. (a) Comparison of two similarity spaces w.r.t. latency. (b) Encrypted deduplication ratio w.r.t. block size in partially decoded and decompressed spaces.

be deduplicated after encryption. However, semantic hashing, while capable of generating similar hashes for similar blocks of data, is not suitable for direct encrypted deduplication.

3.2 A New Similarity Space in Docker Images

Decompressing Docker image layers before deduplication enhances similarity detection and deduplication ratios. However, this process has two drawbacks: (i) re-compression is needed to restore images to their original forms, increasing pull latency, and (ii) decompressing before deduplication reduces system throughput.

Pull latency. To further investigate, we break down the pull latency, which includes downloading and restoring time. Restoring involves fetching chunks and re-compressing using gzip, comprising LZ77 and Huffman encoding. We exclude the fetching time because it is trivial. We perform deduplication on two consecutive versions of the Ubuntu image after decompression. As Figure 4(a) illustrates, LZ77 encoding dominates re-compression time during new version pulls. This raises the question: *Can Docker images be deduplicated after Huffman decoding instead of completely decompression*? Such a method could enable image restoration solely through Huffman encoding, thereby potentially reducing pull latency.

To answer the above question, we partially decode the Docker images using Huffman decode, then deduplicate them after dividing into fixed-size chunks. We assess this method's deduplication ratio against the complete decompression method (including LZ77 and Huffman decoding). Our experiments involve 46 official Ubuntu image versions, totaling 849,347 4KiB blocks in completely decompressed space. Figure 4(b) presents two counter-intuitive results: (i) the state-of-the-art LSH-based MLE technique, particularly using Finesse [87] for block sketch generation, yields a higher deduplication ratio in partially decoded space than in completely decompressed space; (ii) MLE as implemented in AWS Lambda [16] achieves a deduplication ratio over 1 only in completely decompressed space with 4KiB chunking.

We conduct a detailed analysis of deduplication between two continuous Ubuntu images (ubuntu:focal-20230605 and ubuntu:focal-20230624), using the older version's blocks as the base. The results in Figure 5 yield two observations: (i) in the partially decoded space, the layer exhibits more delta bytes compared to the decompressed space; (ii) after decompression, the layer exhibits data bloat, resulting in significantly larger duplicated bytes than in the partially decoded space. Although these duplicated bytes can be removed by deduplicating, storing a duplicate is still necessary. This elucidates the two counter-intuitive findings presented in Figure 4(b): (i) the LSH-based MLE

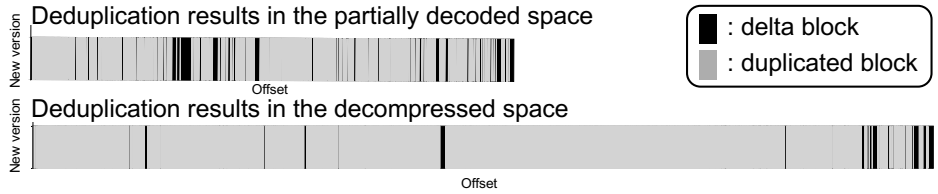


Fig. 5. Deduplication results of LSH-based MLE in partially decoded and completely decompressed spaces for two continuous Docker image versions.

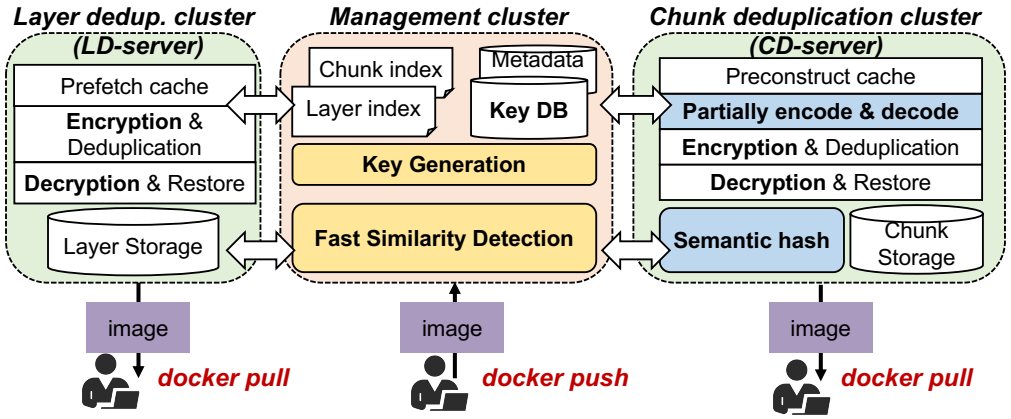


Fig. 6. SimEnc system architecture.

achieves a higher deduplication ratio in the partially decoded space, and (ii) the MLE method is more effective in identifying identical parts at smaller block granularities, as blocks with minor modifications are not amenable to deduplication.

Deduplication throughput. The system faces a trade-off between the deduplication ratio and throughput. Deduplicating all layers in the decompressed space at block granularity maximizes the deduplication ratio but decreases throughput. In contrast, deduplicating at layer granularity in the compressed space enhances throughput but lowers the deduplication ratio. Meanwhile, the pull latency also be compromised.

Previous work [89] has focused on reducing latency by selectively decompressing infrequently accessed layers, at the cost of storage space. For example, DupHunter's selective mode achieves a deduplication ratio of 1.3, while deduplication after decompressing all layers reaches 6.9 [89]. We present a service provider's perspective on whether selective partial decoding of layers based on similarity is feasible. Layers with substantial similarity can be partial decoding followed by deduplication. Conversely, layers with lesser similarity are more suitable for layer-level deduplication. This adaptable approach aims to balance reduced latency, with improved throughput and storage savings.

4 SimEnc Design

In this section, we first provide an overview of SimEnc (§4.1). We then describe in detail how it pre-precesses layers by selecting similarity spaces (§4.2), and how it deduplicates layers by our

novel semantic-aware MLE approach (§4.3). Furthermore, we present the privacy-preserving key generation mechanism (§4.4). Finally, we discuss the SimEnc (§4.5).

4.1 Overview

We propose SimEnc, a high-performance similarity-preserving encryption approach for encrypted Docker image deduplication.

4.1.1 System Architecture. Figure 6 shows the architecture of SimEnc, which consists of two main components: (1) two storage clusters responsible for storing images and pushing layers to clients; and (2) management clusters, which maintain distributed metadata and a key database, and rapidly detect the similarity between clients' pushed layers and existing stored layers.

Management server. The management server serves three main functions: (i) it produces keys for the deduplication process, creating them for the layer deduplication cluster at the layer level and the chunk deduplication cluster at the block level, using the key generation mechanism (cf. §4.3.2); (ii) it manages and stores the keys in the database; and (iii) it deploys our fast similarity space selection mechanism (cf. §4.2) for rapidly detecting layer similarity.

Storage cluster. SimEnc provides two storage clusters to achieve high-performance deduplication. The first cluster is the layer deduplication cluster (LD-cluster) which deduplicates compressed layers at layer granularity. The second cluster is the chunk deduplication cluster (CD-cluster) which contains the unique encrypted chunks in the partially decoded space. It exploits our partial decoding technique to find more identical chunks in the compressed layers and employs partial encoding to restore the original compressed layers. It utilizes our semantic-aware MLE (cf. §4.3) deduplication. SimEnc integrates the prefetch and preconstruct techniques of DupHunter [89] to reduce pull latency.

SimEnc offers three modes to balance the trade-off between deduplication ratio and latency in user pull requests. (1) *Basic deduplication mode n* (B-mode n). For an image with M layers, it performs layer-level encrypted deduplication on the first n layers, using the basic MLE [16]. The remaining $M - n$ layers undergo chunk-level deduplication using our semantic-aware MLE after partial decoding. (2) *High deduplication mode* (H-mode), which deduplicates all layers at chunk level after partial decoding. This process exposes more similarities. (3) *Flexible deduplication mode* (F-mode), utilizes Docker image similarity to select the similarity space for deduplication, balancing deduplication ratio and throughput (cf. §4.2).

Typically, the encryption process is carried out server-side to ensure the confidentiality of container images in the registry. When a user uploads a container image, it is encrypted before storage. This encryption protects against various threats, including unauthorized access and insider threats, by preventing attackers or unauthorized users from accessing the plaintext data of the images, even if they gain access to the storage pool. Encrypted images are stored in the registry but are not executable or accessible without proper authorization. The encryption ensures that, although all users can view the encrypted images, only those with the correct decryption keys can access and use the data. While encrypted images are stored in the registry, they are not executable or accessible without proper authorization. This is achieved by ensuring that the encryption keys are securely managed and that decryption only occurs within the context of authorized users. To enhance security for untrusted cloud environments, SimEnc also integrates the cloud's Trusted Execution Environments (TEEs) for secure key generation and shifts encryption processes to the client side (cf. §4.4). This approach mitigates risks associated with potentially untrustworthy cloud infrastructure, where the underlying hardware or operating system may not be fully secure.

4.1.2 Workflow. Figure 7 illustrates the workflow of SimEnc, featuring two key mechanisms. The fast similarity space selection (§4.2) decides the space—compressed or partially decoded—for

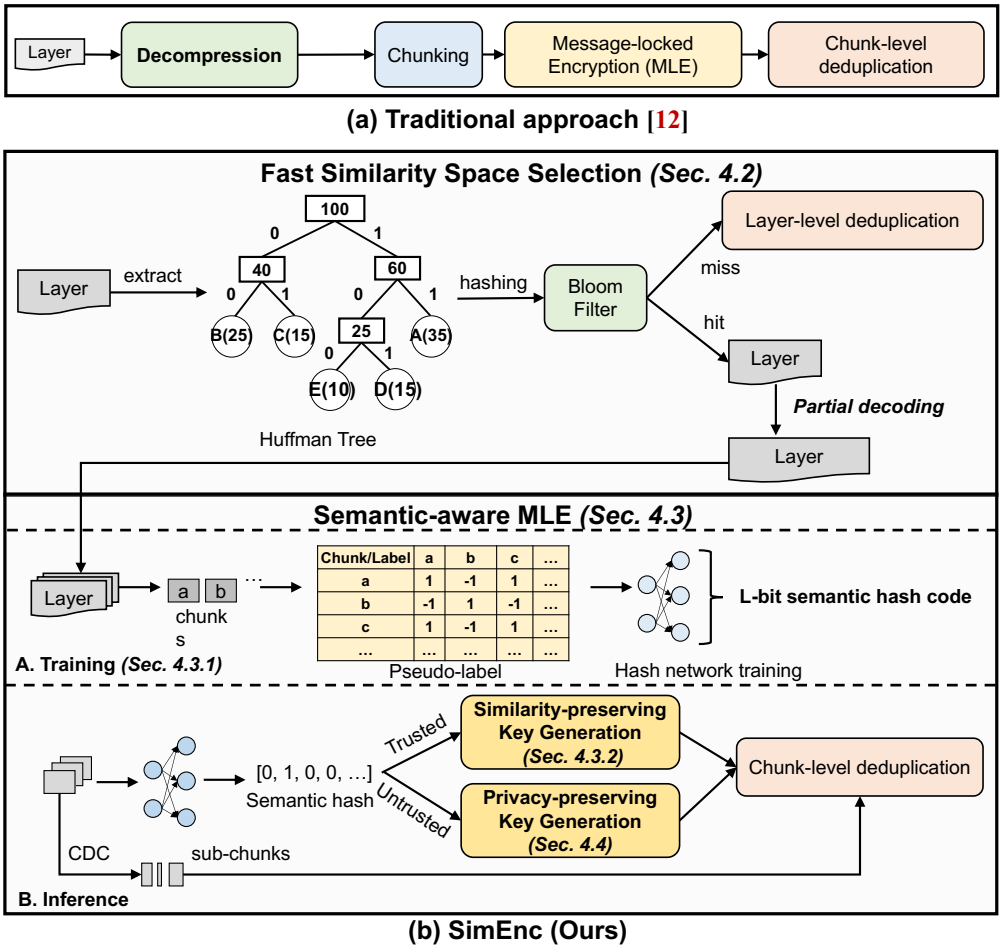


Fig. 7. SimEnc workflow.

encrypted deduplication of each layer. Layers in compressed space undergo basic MLE, while those needing partial decoding are fixed-size chunked for processing with Semantic-aware MLE (§4.3), creating identical hashes for similar blocks. This process enables encrypted deduplication of identical sub-blocks within similar blocks using the same keys.

Fast similarity space selection (§4.2). In F-mode, when the management server receives a new Docker layer, it rapidly determines the deduplication space using Huffman tree similarities. If a similar layer has been partially decoded and deduplicated in the CD-cluster, the new layer is processed there to identify more identical chunks. Otherwise, it's stored in the LD-cluster.

Semantic-aware MLE (§4.3). This process involves two stages: chunk similarity extraction (§4.3.1) and similarity preserving key generation (§4.3.2). Layers are partially decoded and then chunked. For similarity extraction, we use a Hash network to extract semantics from Docker layers, enhancing semantic information through semantic-preserving and similarity contrastive learning. After semantic hash computation, a novel method for similarity-preserving key generation is employed.

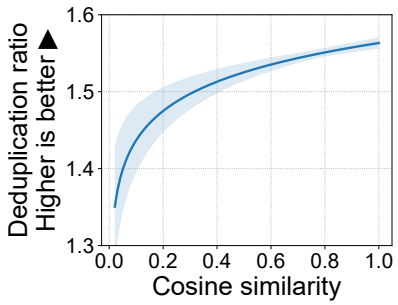


Fig. 8. Dedup. ratio w.r.t. the cosine similarity of pair-wise Huffman trees.

Privacy-preserving key generation (§4.4). To ensure the secure generation of keys within a potentially untrustworthy cloud environment, SimEnc employs the cloud's Trusted Execution Environments (TEEs) [8, 20, 69, 73, 85]. Initially, a secure communication channel is established between the client and the cloud-based TEE. The client then transmits its semantic hash to the TEE. Within this secure environment, the TEE calculates keys by evaluating the hash distances across various client submissions (cf. §4.3.2). After key computation, the TEE dispatches these keys back to the clients. Clients then proceed to encrypt their data locally and subsequently upload this encrypted information to the cloud for deduplication.

4.2 Fast Similarity Space Selection

In F-mode, when a Docker layer is uploaded by a client, SimEnc determines the most suitable space for deduplication. Layer-level deduplication occurs in the compressed space, while Huffman decoding is required for chunk-level deduplication in the partially decoded space. We utilize the Huffman tree in each Docker layer's header to assess layer similarity, as it provides key statistical information about the encoding of compressed content.

A Docker image's Huffman tree is a 286-dimension vector, including 256 ASCII encoding lengths and other data (cf. §2.5). Our intuition is that layers with greater similarity will have more closely aligned Huffman tree statistics. To test this, we evaluate the cosine similarity of Huffman trees across 123,442 layer pairs. The results, shown in Figure 8 with 95% confidence intervals, reveal a positive logarithmic correlation between the deduplication ratio of paired chunks and the cosine similarity of their Huffman trees.

However, the practicality of comparing each layer's Huffman tree in a real-world system poses significant spatial and temporal challenges. We measure that it takes around 5s to compare the cosine similarity of a new layer to the Huffman tree of 10,000 layers stored in the system, which is unacceptable in a high throughput system. Consequently, devising an expedited, efficient method for similarity detection in Docker layers becomes imperative.

To address this challenge, we employ the Bloom filter [12, 13], a compact bit-vector structures representing element sets, allowing for false positives but guaranteeing that unmarked elements are absent. This system maps Huffman trees into bit vectors for rapid comparison. Bloom filter is a space-efficient data structure and is commonly used in high-throughput systems. SimEnc utilizes the Bloom filter to quickly check whether a similar layer belongs to the CD-server.

As depicted in Figure 9, the Bloom filter's bit array size (e.g., 32 bits), is set during system warm-up or update. The details of our algorithm are shown in Algorithm 1. A larger bit array is preferred for lower latency, minimizing false positives and unnecessary deduplication in partially decoded

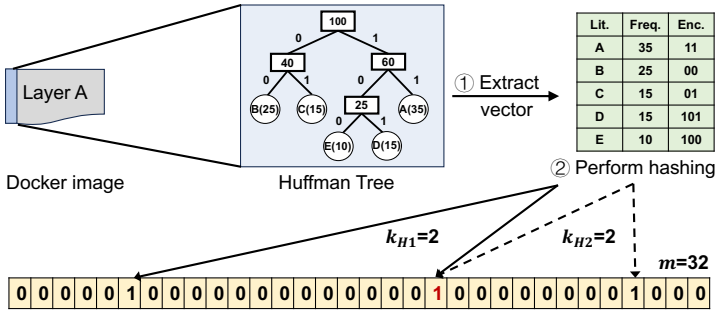


Fig. 9. Fast similarity detection using Bloom Filter [56].

spaces. Conversely, a smaller bit array size increases the likelihood of mapping similar Huffman trees to identical values, improving deduplication after partial decoding. After initialization, the 286-dimension Huffman tree is mapped onto the Bloom filter using multiple hash functions (e.g., Jenkins'hash [14, 42] and Rabin [68]). Layers are deduplicated at the layer level if their hashes are absent in the Bloom filter; if present, they undergo partial decoding for chunk-level deduplication using Huffman decoding. Additionally, by adjusting the size of the Bloom filter and the number of hash functions used, the appropriate false positive rate can be adjusted. During the initialization phase, the Bloom filter starts empty. For example, let's assume that the first three layers uploaded to the system are similar and their hash values of Huffman trees are identical. The hash value of the Huffman tree of the first arriving layer will not hit in the Bloom filter (i.e., layer-level deduplication) because the Bloom filter is empty. But its hash value has been recorded in the Bloom filter. The hash values of the Huffman tree of the second and third arriving layers will hit, so they will be deduplicated at the chunk level.

In the preliminary step of the deduplication process, layers are categorized into three distinct zones based on access frequency. This categorization is performed during the system warm-up phase, where access frequency statistics are collected from the first several user requests. The warm-up phase serves to pre-populate the Bloom filter and assign layers to Hot, Warm, and Cold zones based on their access frequency. *(i) Hot zone*: Layers in the hot zone exhibit the highest access frequency. To minimize access latency, no flattening operations are applied to these layers. *(ii) Warm zone*: Layers in the warm zone have a lower access frequency than the hot zone. These layers undergo partial decoding (i.e., Huffman decoding) to balance deduplication effectiveness and latency. *(iii) Cold zone*: Layers in the cold zone have the lowest access frequency. These layers are fully decompressed (including Huffman and LZ77 decoding), which optimizes deduplication at the cost of higher latency due to the infrequent access. The management server then uses Huffman tree similarities to quickly determine whether a new layer should undergo partial decoding or be processed for deduplication in a different cluster.

4.3 Semantic-aware MLE

The above similarity space selection is the pre-processing of deduplication, we now describe our novel semantic-aware MLE approach for encrypted deduplication. To capture the inherent semantic similarities between Docker layers, we introduce semantic hash [50, 67, 78, 83] into MLE. However, applying semantic hash into MLE is non-trivial. To better understand the problem, we first identify two unique challenges in the context of our scenario as follows.

Algorithm 1: Similarity Space Selection

```
Input  : NewLayer, BloomFilterSize, StoredLayers
Output : Selected similarity space for NewLayer
Initialization: BloomFilter ← InitializeBloomFilter(BloomFilterSize);
while True do
    SHAExists ← CheckSHA(NewLayer, StoredLayers);
    if SHAExists then
        return "Layer already exists. No need to upload.";
    HuffmanTree ← ExtractHuffmanTree(NewLayer);
    HashValues ← ComputeHashes(HuffmanTree);
    IsSimilar ← CheckUpdateBloomFilter(HashValues, BloomFilter);
    if IsSimilar then
        return "Select the partially decoded space";
    else
        return "Select the compressed space.";
```

(C1.) Semantic extraction. Direct application of the semantic hashing technique often leads to biased outcomes, as seen in Figure 19(a), where semantic hashes are unevenly distributed across the hash space. Consequently, training a semantic hash model to achieve uniform data mapping in the hash space presents a significant challenge.

(C2.) Generation identical sketches. While an ideal semantic hashing model is capable of producing similar sketches or hashes for akin data chunks, the MLE framework necessitates identical hashes for similar chunks to enable the encryption of duplicate chunks into identical ciphertexts. However, it is challenging to generate an identical hashed among similar data chunks.

4.3.1 Chunk Semantic Extraction. **Pseudo-label generation.** Recent works show that features extracted from pre-trained deep neural networks contain rich semantic information [34]. However, the extraction of semantic information from image layers remains the following issues: *(i)* to the best of our knowledge, there does not exist a pre-trained model specifically for Docker images to identify semantics; *(ii)* Docker images contain various types of files (e.g., text, binary files, etc.), each with distinct features that are hard to extract and justified similarities. Therefore, our work initially focuses on obtaining the similarity of different file blocks and labeling them correspondingly for training.

In contrast to the conventional cosine distance approach for measuring similarity [78, 83], which often results in high false positive and negative rates at the boundaries of similar chunk clusters [50], our focus is on byte-level rather than semantic-level deduplication. To this end, we first apply random augmentations (i.e., modification with random bytes) to chunks in the partially decoded space. Then, we measure byte-level similarity using the compression ratio metric post delta compression [6, 43, 44, 77, 81]. This addresses the semantic boundary issue by estimating the distance between pairs of blocks through similar block distribution divergence, formulated as [57]:

$$D_{jk}\left(\left\{\mathbf{b}_j^m\right\}_{m=1}^{M},\left\{\mathbf{b}_k^m\right\}_{m=1}^{M}\right)=\frac{1}{M}\sum_{m=1}^{M}\left(\left(\frac{1}{M}\sum_{r=1}^{M}\rho\left(\mathbf{b}_j^m,\mathbf{b}_k^r\right)-\rho\left(\mathbf{b}_j^m,\mathbf{b}_j^r\right)\right)^2+\left(\frac{1}{M}\sum_{r=1}^{M}\rho\left(\mathbf{b}_k^m,\mathbf{b}_k^r\right)-\rho\left(\mathbf{b}_k^m,\mathbf{b}_j^r\right)\right)^2\right),$$

where the $\left\{\mathbf{b}_j^m\right\}_{m=1}^M$, $\left\{\mathbf{b}_k^m\right\}_{m=1}^M$ are the augmented samples of fixed size blocks. $\rho\left(\mathbf{b}_k^m, \mathbf{b}_j^r\right) = 1 - \frac{\text{size}(\Delta(\mathbf{b}_k^m, \mathbf{b}_j^r))}{\text{size}(b_j)}$ is the similarity metric defined by the compression ratio between the blocks, where the $\Delta(\mathbf{b}_k^m, \mathbf{b}_j^r)$ is the delta obtained by utilizing delta compression tools on blocks.

After calculating the distribution distance, we can filter the similar blocks with a specific threshold and generate the pseudo-label for pair-wise blocks which can be constructed as:

$$S_{jk} = \begin{cases} 1 & \text{if } D_{jk} \leq t \\ -1 & \text{if } D_{jk} > t \end{cases}$$

where t is the threshold of distribution distance. We default set the t to 1/2. If the pair is similar, the pseudo-label will be 1; If the pair is dissimilar, the pseudo-label will be -1.

Hash learning network. We follow the existing works to train the hash network for mapping fixed file blocks into fixed length (e.g., 128-bit) hash values. Our deep hash network is based on a convolutional neural network (CNN) architecture followed by a fully-collected layer with L hidden units. The depth of the CNN is determined by the block size of the input. We believe that the larger the block size, the deeper the architecture needs to be constructed to extract the rich semantic features inside the block.

Semantic-preserving learning. The goal of the hash learning network is to map similar blocks into similar hash outputs. We first define the hash similarity function using Hamming distance [64], given by:

$$\hat{S}_{jk} = \frac{1}{L} h_j^T h_k, \quad h_j = sgn(F(b_j; \omega)), \tag{1}$$

where $F(b_j; \omega)$ is L dimension output of our input block data b_j, ω is the learnable parameters of the network, h_j is the corresponding hash codes, $sgn(\cdot)$ is the sign function, and $h_j \in \{-1, 1\}^L$. If a pair of hash codes is similar, the hash similarity function will return a value near 1; If a pair of hash codes is dissimilar, the function will return a value near -1. Then, we design a loss function to minimize the difference between predictive similarity label $\hat{S}_{jk}$ and the pseudo-label S_{jk} of pair-wise blocks, given by:

$$min\mathcal{L}(\omega) = \frac{1}{n^2} \sum_{j=1}^{n} \sum_{k=1}^{n} \left\| S_{jk} - \frac{1}{L} h_j^T h_k \right\|_2^2 \tag{2}$$

Similarity contrastive learning. As we observed in §3.1, existing methods are not efficient at preserving the original similarity of the data. Thus, our insight is that encourage the generation of similar hashes for highly similar chunks and discourage it for less similar ones. To achieve this, we design a contrastive learning loss as follows:

$$min\mathcal{L}(\omega) = \alpha \cdot \mathcal{L}_{sim} + (1 - \alpha) \cdot \mathcal{L}_{dissim} \tag{3}$$

where $\mathcal{L}_{sim}$ and $\mathcal{L}_{dissim}$ are the hash learning loss for similar blocks and dissimilar blocks, respectively, and α is a temperature parameter set to 0.5 as indicated in [19].

4.3.2 Similarity-preserving Key Generation. Semantic hashing is not directly applicable in MLE because it produces similar but not identical hash codes for similar chunks. However, encrypted deduplication requires identical hash codes to serve as or derive the encryption key.

The similarity-preserving key generation in our system leverages clustering, which organizes objects into groups where intra-group relations are closer than those between different groups. Our key idea involves clustering semantic hashes to assign keys to the same class, such as using the representative block key of that class. Unlike other clustering methods like K-means [58],

Algorithm 2: Similarity-preserving Clustering

Input : N file chunks, Semantic hash codes $SH[0, ..., N-1]$ of N chunks
Output: Cluster categories $C[0, ..., N-1]$ for N chunks

```
FeatureMap ← {}, HammingDistances ← {}                          ▷ Initialization
for m = 0 to N − 1 do
    Feature[m] ← LSH(chunk[m])                                  ▷ Obtain features
    FeatureMap[Feature[m]] ← FeatureMap[Feature[m]] ∪ {m}        ▷ Update maps
for feature in keys(FeatureMap) do
    if len(FeatureMap[feature]) > 1 then
        TotalDistance ← 0, PairCount ← 0
        for pair in all pairs of FeatureMap[feature] do
            TotalDistance ← TotalDistance + HammingDistance(SH[pair[0]], SH[pair[1]])
            PairCount ← PairCount + 1
        HammingDistances[feature] ← TotalDistance / PairCount   ▷ Average distance
ϵ ← 90th percentile of values in HammingDistances
C ← DBSCAN(eps=ϵ)                                               ▷ Execute DBSCAN clustering
```

BIRCH [86], and EM-Clustering [84], DBSCAN [72] has several exceptional features in our scenario: *(i)* it forms clusters of arbitrary shapes, doesn't necessitate predefined cluster numbers, and *(ii)* it remains unaffected by the data input order.

In light of this situation, we ask: *is it possible to design an adaptive clustering to set hyperparameters automatically?* If we can, the clustering algorithm can be applied to arbitrary semantic attributes of Docker images as it automatically can extract suitable parameters from a large amount of data. DBSCAN's definition of clusters is based on two parameters: ϵ and $MinPts$. For a point p, the ϵ-neighborhood of p is the set of all the points around p within distance ϵ. The ϵ-neighborhood is formulated as:

$$N_\epsilon(p) = \{q \in D \mid distance(p, q) \leq \epsilon\} \tag{4}$$

If the number of points in the ϵ-neighborhood of ϵ is no smaller than $MinPts$, then all the points in this set, together with p, belong to the same cluster.

To address this issue, our insight is utilizing the LSH method (cf. §3.1) to guide the measurement of semantic hash code distribution, further to adaptive determine hyperparameters. Our intuition is that if the LSH (cf. §2.1) computes identical features for two data blocks, there is a high probability that they are similar blocks. Hence, we can use the feature distribution obtained from identical blocks by LSH to assess the distribution of semantic hash codes derived through semantic hashing.

We propose a similarity-preserving clustering algorithm (Algorithm 2). After computing semantic hashes, we determine the Hamming distances between all file chunk pairs, forming an $N \times N$ matrix for N chunks. Then, we apply LSH to each block to identify representative features, grouping blocks with matching features. We calculate the average Hamming distance within each group.

Figure 10 shows this process's results for 50 Couchbase [25] image versions. Our observations include: *(i)* over 75% of block sets with the same feature are identical (zero Hamming distance); *(ii)* there is a long-tail effect in the CDF distribution. Hence, we set the ϵ value at the 90th percentile of the CDF, adjusting it adaptively for different Docker images. This ϵ hyperparameter, along with the Hamming distance matrix, is input into DBSCAN for final clustering of file blocks.

During system warm-up or updates, the management server aggregates users' chunk semantic hashes, assigning a key to each category (derived from the representative semantic hash). New

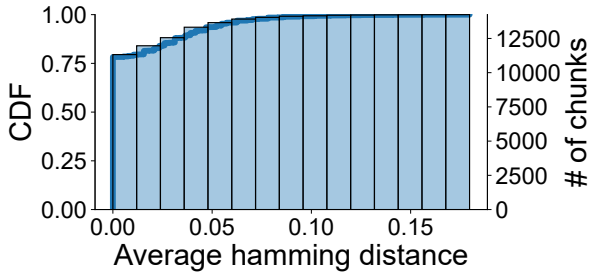


Fig. 10. Statistics of average semantic hash Hamming distance with the same super feature blocks.

uploaded blocks are compared to each category's centroid at online. A block is added to a category if its distance is below a preset threshold, receiving the cluster's key. Otherwise, it forms a new cluster. Increasing the threshold for layers needing strong privacy leads to more distinct classes. After key assignment, chunks are divided into sub-blocks via CDC, encrypted with the key, and then deduplicated by the system.

4.4 Privacy-preserving Key Generation

SimEnc's current encryption methodology is aligned with AWS Lambda's on-demand compute environments [16]. However, the protocol assumes that users are willing to trust a centralized registry for several critical functions: key generation, the encryption of sensitive data, and the secure deduplication of these encrypted data. This assumption predicates the direct uploading of plaintext layers to the registry, which could potentially expose user data to vulnerabilities if the trust assumption is compromised.

Despite typically relying on a trusted cloud environment, SimEnc also accommodates scenarios that lack such trust. In these cases, all encryption-related operations are executed on the client side, introducing a significant overhead due to the necessity of local model inference. The only exception to this local execution is key generation, which uniquely utilizes semantic hash values derived from different users to generate encryption keys. This method ensures that no sensitive user data needs to be exposed to the cloud environment for key generation.

To enhance security in contexts where the cloud cannot be trusted for key generation, SimEnc can employ a cloud-based Trusted Execution Environment (TEE) such as Intel SGX [20]. The process begins with the establishment of a secure communication channel between the client and the cloud-side TEE. Within this secure channel, the client submits its semantic hash, a compact representation of the user's data, which is designed to preserve privacy while allowing for meaningful computation on encrypted values. SimEnc leverages TEE's isolated execution capabilities to securely compute keys by evaluating hash distances between the submissions of various clients. This computation is detailed in Section §4.3.2, where the method for calculating these distances is elaborated. SimEnc mitigates the risk of exposing sensitive computations to potentially malicious cloud operators, as the TEE provides a hardened barrier against both external and internal threats.

After key generation, clients proceed to encrypt their data locally using the keys provided by the TEE. These encrypted data are then uploaded to the cloud, where secure deduplication processes are applied. This strategy ensures that despite the additional computational load imposed on the client's infrastructure, the integrity and confidentiality of user data are maintained throughout the lifecycle of the data in the cloud environment.

4.5 Discussion

Security. SimEnc presents a varient MLE technique for encrypted deduplication. Despite existing studies indicating vulnerability of MLE to brute-force attacks [45], frequency analysis attacks [51, 53], and side-channel attacks [35, 36], it can be defensed against by server-aided MLE [45], proof-of-ownership [35, 52], and server-side deduplication [52, 55, 69], respectively. A practical strategy in AWS Lambda is to mitigate this risk involves varying the salt in the key derivation process [16]. By changing the salt value across different regions and times, the resultant ciphertext also varies. SimEnc can integrate the above methods to enhance security.

Although SimEnc can achieve the same security as AWS Lambda, we still propose a metric score to measure security (# of keys in the system) versus disk savings, as given:

$$\text{Benefit Score} = (\text{Deduplication ratio} - 1)/(\#\text{ of keys})^{\frac{1}{\alpha}}, \tag{5}$$

where $\alpha \in (0, 1]$ is a hyperparameter to regulate whether the system prefers storage saving or security. If α closer to 0, the system prefer a higher deduplication ratio; If $\alpha = 1$, the system only concerns security.

Privacy. SimEnc's key generation process maintains user privacy as it involves comparing semantic hashes from different users on the management server. Due to the inherent properties of hash mapping, a hash code on its own is meaningless and cannot be used to reconstruct the original input [50], thereby safeguarding user data. The implementation of CDC and encryption is carried out separately within each user's space, thereby ensuring that privacy is not compromised.

Encryption procedure. The current encryption process of SimEnc is consistent with AWS Lambda [16]. Considering that external attackers or unauthorized insiders can access the storage pool, SimEnc encrypts images to prevent attackers from accessing the plaintext data. To secure data during transmission, SimEnc employs TLS to establish a secure channel between the client and the server, preventing third-party access to plaintext data. SimEnc typically relies on a trusted cloud, but it can also adapt for scenarios lacking this trust. In such cases, encryption processes are handled on the client side, albeit with increased overhead from local model inference. All operations except for key generation occur locally because it uses semantic hash values from different users to produce keys. To secure key generation in an untrusted cloud, SimEnc could leverage the cloud's Trusted Execution Environment (TEE) [8, 20, 69, 73, 85]. It first establishes a secure communication channel between the client and the cloud-side TEE, and the client submits its semantic hash. The TEE then securely computes keys by calculating hash distances from various clients (cf. §4.3.2) and sends them back. Clients encrypt their data locally and upload the encrypted data to the cloud, where it is deduplicated in the cloud.

Long-term tracking. Given the system's evolving frequent requests and the similarity of layers, it's crucial to monitor the system over time and perform timely rewarming or updates as needed. SimEnc uses a hash network for semantic hashes, the effectiveness hinges on the dataset quality [67].

5 Implementation

We have implemented a prototype of SimEnc in Go by adding ~3,000 lines of code to DupHunter [89]. Due to some libraries in DupHunter original project [88] becoming obsolete or no longer in use, we reconstruct the invalid library references and updated certain libraries to the latest API calls. Our code is open-sourced for public access[4]. We develop partial decoding and encoding tools for Docker layers by ~1500 lines of code in C/C++. During the process of users uploading image files, the system performs partial decoding on the layered data according to the specified mode, segments the generated data, and then stores metadata such as the number and size of file blocks in the

[4] https://github.com/suntong30/SimEnc

main server's memory, to facilitate the restoration and compression of partial data back to its original form. To further enhance the performance of Redis caching, we changed the original Redis singleton connection mode in DupHunter's code to a cluster connection mode and reconstructed all API calls for Redis memory operations. In addition, we utilize the FastCDC [82] as the CDC implementation and exploit AES-CTR [70] for encryption.

In training the semantic hash, we employ a CNN architecture [49] as the semantic hashing model [71]. Specifically, for 512KiB input chunks, our model comprises eight convolutional (conv) layers, with each conv layer being followed by ReLU, BatchNorm, and MaxPool layers. Subsequent to the CNN processing, we deploy two linear layers to generate the hash codes. Note that the neural network architecture is specific to the input chunk size. The greater the size of the input chunk, the deeper the network structure required to extract additional semantic information, necessitating a larger number of Linear parameters. The training process generates a semantic hash model to extract semantic information for each chunk. This information is used for chunk-level deduplication of encrypted images. SimEnc collects the public images from Docker Hub to create a warm-up dataset and separate it into a training set and a test set (cf. §6). Then, the model is initialized with random parameters and trained with pseudo labels derived from the delta compression algorithm (cf. §4.3.1), using the stochastic gradient descent algorithm to minimize the loss function. Once trained and validated against the test set for accuracy, the model is deployed as the online inference model.

6 Evaluation

6.1 Methodology

Evaluation platform. We set up SimEnc on three PC servers, each equipped with a 20-core Intel i9-10900K CPU (@3.70 GHz), 128GB DDR4 DRAM, and a 4TB S690MQ SSD. All servers run Ubuntu 20.04 as their operating system and are interconnected via a 800Mbps network. We use one GeForce RTX 3090 Ti for training and inference processes of the semantic hashing network. For each experiment, we conduct ten runs to calculate the average value.

Baselines. We compare SimEnc against three baselines.

- DupHunter [89, 90], the state-of-the-art Docker registry for plaintext deduplicaiton. We reproduce DupHunter's code [88] on GitHub with the deduplication, restoring, caching, and preconstructing layers mechanisms mentioned in [89]. We configure the cache size as 5% of total size of unique layers in the workload, and utilize the LRU [65] cache algorithm for caching.
- AWS Lambda registry [16], the state-of-the-art serverless platform for encrypted Docker image deduplication using MLE. We adhere to the settings outlined in [16], which include setting a fixed block size of 512KiB, using the SHA256 hash of the block as the key, and encrypting with AES.
- Improved AWS Lambda. We integrate LSH-based MLE in AWS Lambda with Finesse [87], to generate identical keys for similar chunks. We use twelve (3×4) Rabin fingerprint functions with a window size of 48 bytes in total. We set the max, average, and min chunk size of CDC to 1KiB, 0.5KiB, and 0.2KiB [5]. In addition, a chunk may have multiple similar chunks, and we select the first matched chunk as its base, which is also known as "FirstFit" [48].

Datasets and workloads. Table 2 summarizes the characteristics of our datasets and workloads in terms of the size and unique layers. Our dataset comprises sequential version images downloaded from DockerHub, selected for two reasons: *(i)* they are popular images in real-world applications, widely used for reuse purposes (e.g., Ubuntu [28] of operating systems and Couchbase [25] of databases), and have been studied in previous research [85]; *(ii)* as they are sequential versions, some files within the compressed layers have been modified, making it unlikely to find duplicates at

Table 2. Summary of the evaluated datasets and workloads.

Dataset/Workload	#Layer	#Unique Layer	Comp. size	Partially decoded size	Decomp. size
Ubuntu [28]	46	46	1.18 GiB	1.67 GiB	3.24 GiB
Couchbase [25]	516	263	17.74 GiB	35.85 GiB	41.29 GiB
IBM (Dal) [39]	2000	758	11.23 GiB	15.36 GiB	28.97 GiB
IBM (Fra) [39]	2000	700	10.77 GiB	14.57 GiB	27.88 GiB
IBM (Lon) [39]	2000	710	9.49 GiB	13.11GiB	25.11 GiB
IBM (Syd) [39]	2000	503	19.01 GiB	25.73 GiB	48.48 GiB
IBM (Random) [39]	13619	7521	263.13 GiB	318.8GiB	643.95 GiB

the layer level, thus facilitating our research. Our workload involves IBM's trace dataset [9, 39]. To evaluate DupHunter's performance with production registry workloads, we utilize IBM traces from four production registry clusters (Dal, Fra, Lon, and Syd) [9, 39, 89], covering approximately 80 days. We employ the Docker registry trace replayer [39] to replay valid requests from each workload. For each workload, we use the first 5,000 requests to warm up the system. We modify the replayer to align requested layers in the IBM trace with actual layers downloaded from Docker Hub [24], based on layer size. As a result, each layer request involved pulling or pushing an actual layer. For manifest requests, we generated random, well-formed manifest files, following DupHunter [89]. For each IBM workload, we use the traces uploaded by the first 5000 user requests in each workload to warm up the system. Specifically, we first match the image layers in the trace that have the same size as the image layer we downloaded from Docker Hub. These layers will be used for the warm-up of the workload experiment. Then, we upload them from the client to the Docker registry for registration. We extract the Rabin hash value of the Huffman tree of each layer in the management server, and use the Bloom filter to mark whether to perform layer level or chunk level deduplication. After all 5000 traces are processed, all layers marked as layer level deduplication will be deduplicated by MLE in the compressed space, and all layers marked as chunk level deduplication. To the latter, we first partially decode them, perform 512KiB fixed chunking with pending, and use the above-trained model to generate semantic hash values. Then, their semantic hash values are clustered through the similarity-preserving key generation mechanism (cf. Sec. 4.3.2) and the corresponding keys are generated. Finally, they are divided into sub-chunks using the CDC algorithm, and the sub-chunks are AES encrypted using the corresponding key. The keys and chunks' metadata will be safely stored in the management cluster.

Warm-up. The warm-up process can be divided into three stages: (1) model setup and training, (2) deduplication cluster warm-up (including initial layer ingestion and bootstrapping), and (3) system rewarming. To prepopulate the deduplication cluster, we collect traces and corresponding layers from the first several user requests and filling the Bloom filter to perform initial layer ingestion. It calculates the hash of the Huffman tree for each layer at the management server and utilizes a Bloom filter to decide whether to apply layer-level or chunk-level deduplication. Once all requests are processed, layers designated for layer-level deduplication undergo deduplication by MLE in compressed space, while others are partially decoded and chunked. The trained model then generates semantic hash values for each chunk, which are clustered using the similarity-preserving key generation mechanism (cf. §4.3.2) to produce keys. These chunks are divided into sub-chunks via the CDC algorithm and encrypted. The keys and metadata for the chunks are securely stored in the management cluster. The primary rewarming involves updating the semantic hash network and the deduplication cluster. A comprehensive but resource-intensive method is to reset all layers to their initial state, retrain the semantic hash network, and refresh the deduplication cluster.

System rewarming. Since the characteristics of images uploaded by users will change over time, SimEnc periodically needs to rewarm its system for enhanced performance. The primary rewarming

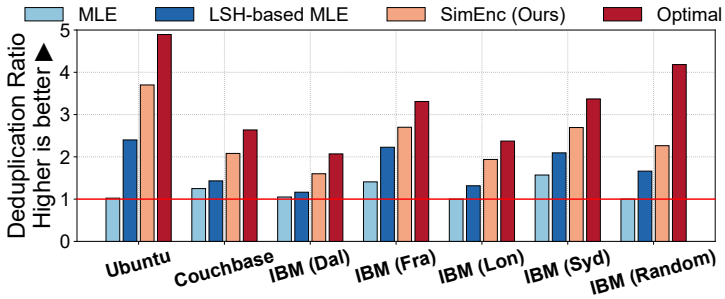


Fig. 11. Deduplication ratio in partially decoded space.

involves updating the semantic hash network and the deduplication cluster. A comprehensive but resource-intensive method is to reset all layers to their initial state, retrain the semantic hash network, and refresh the deduplication cluster. Alternatively, SimEnc employs an efficient incremental rewarming approach: (1) It uses newly uploaded layers to continuously train the model, allowing it to adapt to the current semantic distribution; (2) It monitors the distribution of layer similarity and popularity, and selectively updates the deduplication cluster manually.

6.2 Deduplication Ratio

Deduplication ratio in partially decoded space. To demonstrate the deduplication ratio of SimEnc, we conduct all layers of deduplication in the partially decoded similarity space. Each layer is divided into 512KiB chunks after partial decoding. The results are shown in Figure 11. We observe that SimEnc achieves the highest deduplication ratio in tested datasets and workloads. In two datasets and five workloads, SimEnc achieves an average deduplication ratio that is 38.6% higher than the LSH-based MLE (enhanced for AWS Lambda [16]) and 109.2% higher on average compared to the MLE implemented in AWS Lambda [16]. Specifically, SimEnc outperforms LSH-based MLE by up to 54.2% and MLE by up to 261.7% in the Ubuntu dataset. We perform fine-grained statistics on deduplicated blocks on the Ubuntu dataset. We observe that compared with brute force search, SimEnc can identify 93% of data block similarities through semantic-aware MLE. The MLE method suffers from high perturbation and can only identify identical blocks (occupying 30.1% of total blocks). Although LSH-based MLE can generate the same key for similar blocks through super features, it still has difficulty coping with the incremental modifications that occur at the feature extraction point, and thus can only identify 68.0% of similar blocks.

Deduplication ratio vs. latency. We evaluate SimEnc's deduplication ratio and pull latency trade-off with different deduplication modes (cf. §4). We replay the four production workloads [9, 39] and record the average `pull` layer latency. The results are illustrated in Table 3. In B-mode n, the deduplication ratio diminishes as n increases. Conversely, relative to B-mode 1, the average latency escalates to 1.0x, 0.73x, and 0.54x in B-mode 1, 2, 3, respectively. This latency reduction is due to the decreased number of layers subject to deduplication following partial decoding, which is proportional to the increment in n. While this reveals greater similarities, thus enhancing the deduplication ratio, it concurrently incurs added time overhead from the increased partially encoding operations during user requests.

We now discuss H-mode and F-mode. H-mode achieves the highest deduplication ratio among all four production workloads, a result of deduplicating all compressed layers in the partially decoded similarity space. However, this leads to the highest latency costs. In F-mode, SimEnc employs the fast similarity space selection mechanism (cf. §4.2). Here, layers are selectively deduplicated in

Table 3. Deduplication ratio vs. pull layer latency.

Mode	Deduplication ratio				Latency (compared to B-mode 1)			
	Workload							
	Dal	Fra	Lon	Syd	Dal	Fra	Lon	Syd
B-mode 1	1.28	2.68	1.65	1.87	1.0x	1.0x	1.0x	1.0x
B-mode 2	1.24	2.60	1.58	1.72	0.94x	0.75x	0.70x	0.53x
B-mode 3	1.21	2.40	1.51	1.65	0.62x	0.61x	0.52x	0.42x
H-mode	1.60	2.71	1.94	2.69	1.44x	1.05x	1.57x	1.09x
F-mode	1.55	2.71	1.81	2.62	1.28x	0.87x	1.42x	1.07x

Table 4. Comparison of deduplication ratio and average pull layer latency on IBM traces [9, 39].

High deduplication mode (H-mode)		
Docker Registry	Deduplication ratio	Latency (s)
DupHunter [89]	1.866	0.285
SimEnc (Ours)	2.710	0.206
Flexible mode (F-mode)		
Docker Registry	Deduplication ratio	Latency (s)
DupHunter [89]	1.45	0.124
SimEnc with DupHunter'selective method	1.49	0.117
SimEnc (Ours)	2.70	0.127

the partially decoded space at chunk granularity. Consequently, F-mode positions itself between B-mode 1 and H-mode, striking a balance with a deduplication ratio nearing that of H-mode, yet maintaining a latency comparable to B-mode 1.

Comparison with DupHunter. We compare SimEnc with the DupHunter [89] in terms of deduplication ratio and pull latency under the IBM (Fra) workload. Note that the Duphutner deduplicates plaintexts of Docker images while SimEnc deduplicates encrypted images. In the H-mode, we configure all layers to be partially decoded and completely decompressed before deduplication for SimEnc and DupHunter, respectively. In the F-mode, DupHunter utilizes selective decompression according to the layer popularity [89], while SimEnc deploys our fast similarity space selection mechanism.

The comparative results are shown in Table 4. *(i)* In H-mode, SimEnc achieves a 45.2% higher deduplication ratio and a 27.7% lower latency than DupHunter. Despite DupHunter's approach of deduplicating layers in plaintext after complete decompression at file granularity, SimEnc operates at block granularity. SimEnc encrypts layers using our semantic-aware MLE after partial decoding, leading to a superior deduplication ratio compared to DupHunter's plaintext method, even though SimEnc deduplicates ciphertext. Additionally, the partial encoding time required by SimEnc during restoration is shorter than DupHunter's recompression with `gzip`. Furthermore, while SimEnc necessitates decrypting the encrypted blocks during restoration, this process averages only 0.05s, counteracted by the time saved between partial encoding and `gzip` compression. *(ii)* In F-mode, DupHunter implements selective decompression for layer deduplication based on layer popularity, achieving a 56.5% reduction in pull latency compared to H-mode, but at the expense of a 22.3% decrease in deduplication ratio. Similarly, SimEnc, adopting DupHunter's flexible strategy reduces latency by 43.2% while also reducing the deduplication rate by 45.2% compared to H-mode.

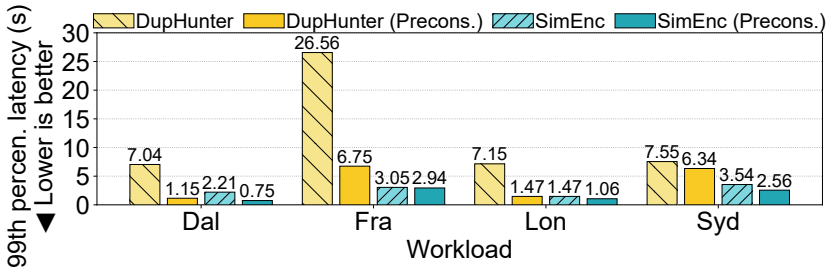


Fig. 12. 99^{th} percentile pull layer latency.

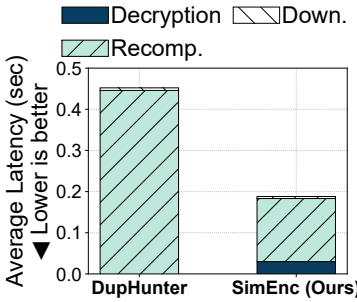


Fig. 13. Pull latency breakdown.

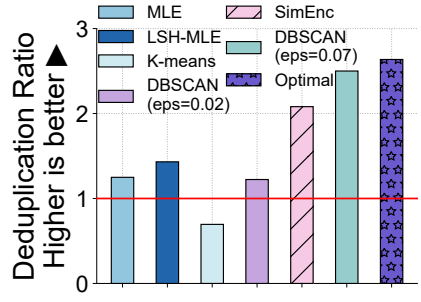


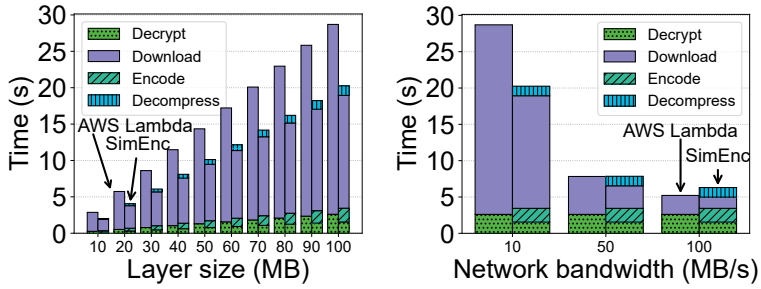
Fig. 14. Deduplication ratio w.r.t. different hyperparameters and clustering algorithms.

SimEnc leverages our fast similarity space selection mechanism (cf. §4.2), enhancing the deduplication ratio by 86.2% over DupHunter while maintaining comparable latency. This results in a modest 2.4% increase in latency overhead relative to DupHunter.

6.3 Latency

Overall pull latency. Figure 12 displays the 99th percentile latency of SimEnc. We observe that compared to DupHunter [89], SimEnc achieves an average latency reduction of 72.39% across four workloads. Notably, in the Fra workload, SimEnc's 99^{th} percentile pull latency is reduced by up to 88.53%. This improvement is due to our deduplication in the partially decoded space, while DupHunter performs deduplication in the completely decompressed space, requiring both Huffman and LZ77 encoding processes for restoration. In contrast, SimEnc performs deduplication in partially decoded space, which only necessitates Huffman encoding in restoration. SimEnc performs better on the 'Fra' workload compared to other workloads than DupHunter because DupHunter without preconstruction has the longest pull latency. Compared to other workloads, 'Fra' has trace data with very large size layers. Without using the preconstruction mechanism, requesting these layers would require recompressing them from an uncompressed state, which would consume a lot of time. However, preconstruction allows these layers to be restored and recompressed in advance, thus, SimEnc achieves the greatest improvement over DupHunter in 'Fra'. Interestingly, our findings show that even with the preconstruct cache mechanism active, SimEnc maintains superior latency performance compared to DupHunter. This is attributed to the fact that while the preconstruct cache can anticipate and pre-restore the subsequent layer, it is still constrained by a bottleneck effect. Consequently, in the most favorable scenario, the longest time taken for a pull request is dictated by the restoration time of the layer with the highest byte count.

Pull latency breakdown. We break down the pull latency of DupHunter [89] and SimEnc under the IBM (random) workload. We make two main observations from Figure 13. (i) The average



(a) Impact of different layer sizes on latency at 10MB/s network bandwidth.

(b) Impact of different bandwidths on latency for processing 100MB layer size.

Fig. 15. Comparison of end-to-end latency under different layer sizes and network bandwidth.

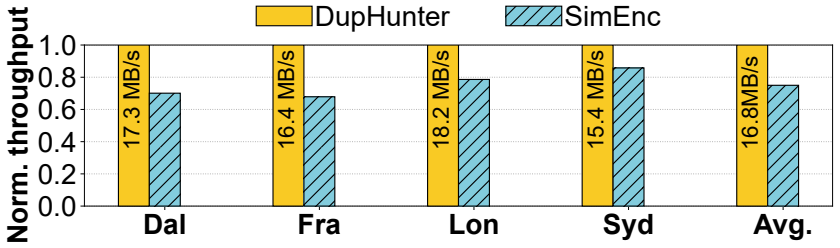


Fig. 16. Deduplication throughput.

latency of SimEnc is 58.4% lower than DupHunter. (ii) Re-compression (re-encoding in SimEnc) time accounts for 98.6% and 81.4% of the average time of DupHunter and SimEnc, respectively. This suggests that SimEnc is better than existing methods in terms of latency because existing methods require recompression, while SimEnc only requires Huffman encoding.

Comparison of latency with AWS Lambda. Despite AWS Lambda [16] is a serverless platform where client images don't require recompression after restoration (as they can be directly mounted and executed), a fair comparison with SimEnc is possible in terms of the end-to-end latency from requesting to starting the Docker image. The end-to-end latency for AWS Lambda primarily comprises decryption and downloading [16], whereas for SimEnc, it includes decryption, partial encoding, downloading, and decompression. We evaluate the impact of different network bandwidths and layer sizes on end-to-end latency. Figure 15 shows the results.

In low-bandwidth (<50MB/s) scenarios, SimEnc achieves lower end-to-end latency compared to AWS Lambda because it transmits original compressed data, whereas AWS Lambda transmits flattened data. Additionally, with large file sizes, SimEnc maintains lower latency. Despite needing re-encoding and decompression, this process is faster than AWS Lambda's transmission of 2-3 times more data.

6.4 Throughput

Figure 16 shows the average deduplication throughput of SimEnc and DupHunter under different workloads, normalized to DupHunter. SimEnc provides up to 85.8% (75.6% on average across all workloads) of the average throughput of DupHunter. To better understand the performance overheads of SimEnc, we measure the average throughput of each step per input data block during the encrypted deduplication process. We find that the performance overhead is mainly due to the

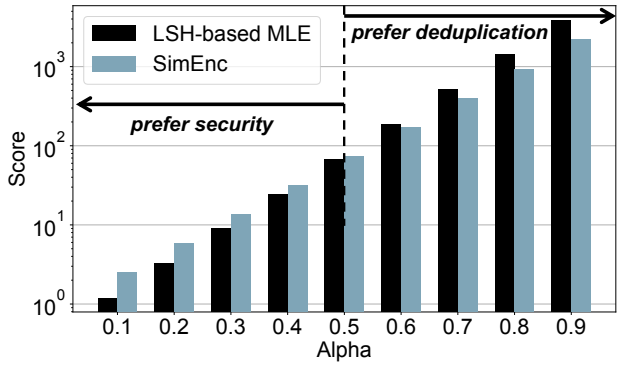


Fig. 17. Benefit score w.r.t. different alpha settings.

semantic-aware MLE in encrypted deduplication. Our measurements indicate that SimEnc achieves an average throughput of 135.2MB/s when partially decoding a layer. Utilizing the LSH-based MLE method for deduplication, this throughput averages 43.7MB/s. However, when employing a semantic hashing model to generate data chunk sketches, the throughput decreases to 16.8MB/s, turning it into a bottleneck. We note that SimEnc currently relies on a single GPU for inference. Utilizing multiple GPUs for parallel inference could improve throughput, potentially enabling SimEnc to outperform DupHunter.

6.5 Semantic-aware MLE Effectiveness

Clustering effectiveness. To evaluate the effectiveness of the similarity-preserving clustering algorithm (cf. §4.3), we compare it with different clustering algorithms and hyperparameters. We manually set different ϵ hyperparameters for DBSCAN in our semantic-aware MLE, and also replace the clustering algorithm with the K-Means algorithm (K=100).

The results are shown in Figure 14, revealing the following: (i) utilizing the K-Means algorithm for clustering semantic hashes prior to encrypted deduplication results in the lowest deduplication ratio, even producing negative storage saving benefits. This outcome is primarily due to K-Means' suitability for spherical data and its effectiveness in clustering similar data in Euclidean space. In contrast, our semantic hashing deals with arbitrarily shaped high-dimensional data, with similarity being defined in Hamming space, making K-Means less effective in this context. (ii) Using our novel similarity-preserving clustering algorithm, SimEnc adaptively set the ϵ at 0.3 in this case. Our deduplication ratio increased by 33.3% compared to LSH-based MLE and by 66.7% compared to MLE (deployed in AWS Lambda). This is due to SimEnc's ability to assign the same key to similar data and perform fine-grained encrypted deduplication on sub-blocks, thereby achieving more storage savings. (iii) As the ϵ hyperparameter of DBSCAN increases, the deduplication ratio also becomes higher. This is because ϵ determines the class distance, and the larger the ϵ, the more likely it is to cluster data from farther distances together. However, this can pose significant security risks. For example, when ϵ is set to 0.7, although its deduplication ratio is close to optimal, it generates only 3 unique keys for 73,406 512KiB blocks.

We now use the benefit score (cf. §4.5) to measure the security and storage savings. In the above case, LSH-based MLE and SimEnc achieve the deduplication ratio of 1.43 and 2.08, respectively, using 25,032 and 4,761 unique keys. We show the benefit score of Couchbase dataset in Figure 17. When $\alpha \leq 0.5$ (indicating a preference for deduplication over security), SimEnc surpasses LSH-based MLE in performance. For α greater than 0.6, where security is paramount, LSH-based MLE

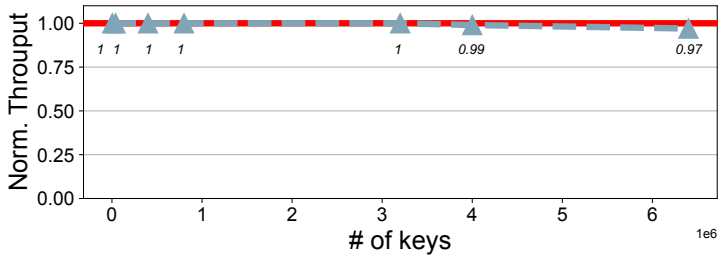


Fig. 18. Comparison of throughput in privacy-preserving key generation w.r.t. # of keys (normalized throughput of native approach = 1).

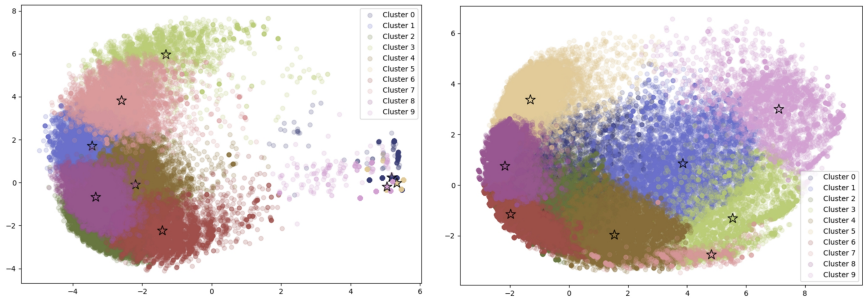
is more appropriate. It's worth noting that this is based on the SimEnc prototype. For enhanced privacy, the similarity-preserving key generation in SimEnc can be modified to favor the generation of unique keys for each chunk.

Privacy-preserving key generation effectiveness. To show the effectiveness of our privacy-preserving key generation mechanism, we evaluate the relative throughput of different key counts compared to direct perform semantic hashing comparison. In our system, Intel SGX serves as the Trusted Execution Environment (TEE), featuring 128MB of secure memory, of which approximately 96MB is available [47, 62]. We configure a cluster of keys within the TEE, and users compute the semantic hash locally. Following this computation, the hash value is securely transmitted to the TEE of SimEnc, as outlined in §4.4. As shown in Figure 18, the TEE conducts semantic hash comparisons. When the cluster contains fewer than 3,200,000 keys, the privacy-preserving key generation mechanism incurs no performance penalty due to the native execution capabilities of the TEE. However, as the number of keys increases to 6,400,000, a performance degradation of 3% occurs. This degradation is attributed to the secure memory exceeding 96MB, necessitating the swapping of pages between the protected enclave memory and the regular, unprotected memory. Such operations require additional encryption and decryption processes. Nonetheless, this performance loss can be alleviated by employing a distributed TEE cluster.

Chunk semantic extraction effectiveness. To evaluate the effectiveness of our chunk semantic extraction (cf. §4.3.1), we trained two hashing networks with identical architecture, one utilizing contrastive learning and the other without it. Both networks underwent training on the same dataset, employing identical learning rates and training epochs. Upon completion of the training phase, these networks were utilized to perform inference on 110,120 512KiB data chunks, to derive their respective semantic hash values. Subsequently, we apply the same DBSCAN parameters for clustering and utilize PCA [59] to condense the dimensionality of the high-dimensional semantic hashes to 2 dimensions for a more comprehensible analysis. Figure 19 presents the visualization of semantic hash codes. Figure19(a) displays a bias with clustering on the left, due to the absence of contrastive learning in the model, making slightly similar data appear very similar in hash space. Conversely, Figure19(b), employing contrastive learning, shows an even distribution of hashes, highlighting the effectiveness of SimEnc's chunk semantic extraction method.

7 Conclusion

SimEnc realizes a high-performance similarity-preserving encryption approach for deduplication of encrypted Docker images. It is the first work deduplicating encrypted layers in the partially decoded space, where can achieve better deduplication ratio, latency, and throughput. It first employs the



(a) without (b) with

Fig. 19. The visualization of clustering in semantic hash code space w/wo hashing contrastive learning.

semantic hash technique in MLE to overcome the limitations of existing MLE approaches. We show that SimEnc outperforms existing approach in performance and storage savings.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions. This work is supported by the National Natural Science Foundation of China under Grant No. 62272407, and 62302096, the "Pioneer" and "Leading Goose" R&D Program of Zhejiang under grant No. 2023C01033, the National Youth Talent Support Program. Yi Gao and Wei Dong are the corresponding authors.

References

[1] 2024. containerd-imgcrypt. https://github.com/containerd/imgcrypt.
[2] 2024. cosign. https://github.com/sigstore/cosign.
[3] 2024. Docker Content Trust (DCT). https://docs.docker.com/engine/security/trust.
[4] 2024. Dockerfile reference. https://docs.docker.com/reference/dockerfile/.
[5] 2024. fastcdc 1.5.0. https://pypi.org/project/fastcdc.
[6] 2024. HDiffPatch. https://github.com/sisong/HDiffPatch.
[7] Atul Adya, Bill Bolosky, Miguel Castro, Ronnie Chaiken, Gerald Cermak, John JD Douceur, Jon Howell, Jay Lorch, Marvin Theimer, and Roger Wattenhofer. 2002. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of USENIX OSDI*.
[8] Tiago Alves. 2004. TrustZone: Integrated Hardware and Software Security. *Information Quarterly* 3 (2004), 18–24.
[9] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S Warke, Heiko Ludwig, et al. 2018. Improving Docker Registry Design based on Production Workload Analysis. In *Proc. of USENIX FAST*.
[10] AWS. [n. d.]. What is RESTful API? https://aws.amazon.com/what-is/restful-api/.
[11] AWS. 2024. AWS Lambda. https://aws.amazon.com/lambda/?nc1=h_ls.
[12] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.
[13] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrah, Sushil Singh, and George Varghese. 2006. Beyond Bloom filters: From approximate membership checks to approximate state machines. *ACM SIGCOMM computer communication review* 36, 4 (2006), 315–326.
[14] Alex D Breslow, Dong Ping Zhang, Joseph L Greathouse, Nuwan Jayasena, and Dean M Tullsen. 2016. Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing. In *Proc. of USENIX ATC*.
[15] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. 1998. Min-wise independent permutations. In *Proc. ACM STOC*.
[16] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. 2023. On-demand Container Loading in AWS Lambda. In *Proc. of USENIX ATC*.

[17] Emiliano Casalicchio and Stefano Iannucci. 2020. The state-of-the-art in Container Technologies: Application, Orchestration and Security. *Concurrency and Computation: Practice and Experience* 32, 17 (2020), e5668.
[18] Doron Chen, Michael Factor, Danny Harnik, Ronen Kat, and Eliad Tsfadia. 2021. Length preserving compression: Marrying encryption with compression. In *Proc. of ACM SYSTOR*.
[19] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *ICML*. 1597–1607.
[20] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016).
[21] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. 2002. Pastiche: Making Backup Cheap and Easy. In *Proc. of USENIX OSDI*.
[22] Peter Deutsch. 1996. Rfc1951: Deflate compressed data format specification version 1.3.
[23] IBM Developer. 2024. Encrypted container images for container image security at rest. https://developer.ibm.com/articles/encrypted-container-images-for-container-image-security-at-rest.
[24] Docker. [n. d.]. Docker Hub Container Image Library. https://hub.docker.com/.
[25] Docker. 2024. Couchbase (Docker Official Image). https://hub.docker.com/_/couchbase.
[26] Docker. 2024. Docker Registry. https://github.com/distribution/distribution.
[27] Docker. 2024. Docker Registry HTTP API V2. https://distribution.github.io/distribution/spec/api.
[28] Docker. 2024. Ubuntu (Docker Official Image). https://hub.docker.com/_/ubuntu.
[29] John R Douceur, Atul Adya, William J Bolosky, P Simon, and Marvin Theimer. 2002. Reclaiming Space from Duplicate Files in a Serverless Distributed File System. In *Proc. of IEEE ICDCS*.
[30] Jean-loup Gailly and Mark Adler. 1992. GNU gzip. *GNU Operating System* (1992).
[31] Chuang Gan, Yuchong Hu, Leyan Zhao, Xin Zhao, Pengyu Gong, Wenhao Zhang, Lin Wang, and Dan Feng. 2023. Enabling Encrypted Delta Compression for Outsourced Storage Systems via Preserving Similarity. In *Proc. of IEEE ICCD*.
[32] Ruihao Gao, Xueqi Li, Yewen Li, Xun Wang, and Guangming Tan. 2022. MetaZip: a high-throughput and efficient accelerator for DEFLATE. In *Proc. of ACM/IEEE DAC*. 319–324.
[33] Ioannis Giannakopoulos, Konstantinos Papazafeiropoulos, Katerina Doka, and Nectarios Koziris. 2017. Isolation in Docker through Layer Encryption. In *Proc. of IEEE ICDCS*.
[34] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proc. of IEEE/CVF CVPR*.
[35] Shai Halevi, Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. 2011. Proofs of ownership in remote storage systems. In *Proc. of ACM CCS*.
[36] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. 2010. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy* (2010).
[37] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers. In *Proc. of USENIX FAST*.
[38] David A Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proc. of IEEE IRE* 40, 9 (1952), 1098–1101.
[39] IBM. 2024. Docker Registry Trace Player. https://dssl.cs.vt.edu/drtp/.
[40] IBM. 2024. IBM Cloud. https://www.ibm.com/cloud.
[41] Piotr Indyk, Rajeev Motwani, Prabhakar Raghavan, and Santosh Vempala. 1997. Locality-preserving hashing in multidimensional spaces. In *Proc. of ACM STOC*.
[42] B Jenkins. [n. d.]. 4-byte Integer Hashing. http://burtleburtle.net/bob/hash/integer.html.
[43] Lewei Jin, Wei Dong, Bowen Jiang, Tong Sun, and Yi Gao. 2024. Exploiting Multiple Similarity Spaces for Efficient and Flexible Incremental Update of Mobile Apps. In *Proc. of IEEE INFOCOM*.
[44] J.Macdonald. [n. d.]. xdelta3. http://xdelta.org.
[45] Sriram Keelveedhi, Mihir Bellare, and Thomas Ristenpart. 2013. DupLESS: Server-Aided Encryption for Deduplicated Storage. In *Proc. of USENIX Security*.
[46] John Kelsey. 2002. Compression and information leakage of plaintext. In *International Workshop on Fast Software Encryption*. Springer, 263–276.
[47] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. Shieldstore: Shielded in-memory key-value storage with sgx. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–15.
[48] Purushottam Kulkarni, Fred Douglis, Jason D LaVoie, and John M Tracey. 2004. Redundancy Elimination within Large Collections of Files.. In *Proc. of USENIX ATC*.
[49] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
[50] Huining Li, Xiaoye Qian, Ruokai Ma, Chenhan Xu, Zhengxiong Li, Dongmei Li, Feng Lin, Ming-Chun Huang, and Wenyao Xu. 2023. TherapyPal: Towards a Privacy-Preserving Companion Diagnostic Tool based on Digital Symptomatic

Phenotyping. In *Proc. of ACM MobiCom*.
[51] Jingwei Li, Chuan Qin, Patrick PC Lee, and Xiaosong Zhang. 2017. Information leakage in encrypted deduplication via frequency analysis. In *Proc. of IEEE/IFIP DSN*.
[52] Jingwei Li, Yanjing Ren, Patrick PC Lee, Yuyu Wang, Ting Chen, and Xiaosong Zhang. 2023. FeatureSpy: Detecting Learning-Content Attacks via Feature Inspection in Secure Deduplicated Storage. In *Proc. of IEEE INFOCOM*.
[53] Jingwei Li, Guoli Wei, Jiacheng Liang, Yanjing Ren, Patrick PC Lee, and Xiaosong Zhang. 2022. Revisiting frequency analysis against encrypted deduplication via statistical distribution. In *Proc. of IEEE INFOCOM*.
[54] Jingwei Li, Zuoru Yang, Yanjing Ren, Patrick PC Lee, and Xiaosong Zhang. 2020. Balancing Storage Efficiency and Data Confidentiality with Tunable Encrypted Deduplication. In *Proc. of ACM EuroSys*.
[55] Mingqiang Li, Chuan Qin, and Patrick PC Lee. 2015. CDStore: Toward Reliable, Secure, and Cost-Efficient Cloud Storage via Convergent Dispersal. In *Proc. of USENIX ATC*.
[56] Lailong Luo, Deke Guo, Richard TB Ma, Ori Rottenstreich, and Xueshan Luo. 2018. Optimizing bloom filter: Challenges, solutions, and comparisons. *IEEE Communications Surveys & Tutorials* 21, 2 (2018), 1912–1949.
[57] Xiao Luo, Daqing Wu, Zeyu Ma, Chong Chen, Minghua Deng, Jianqiang Huang, and Xian-Sheng Hua. 2021. A Statistical Approach to Mining Semantic Similarity for Deep Unsupervised Hashing. In *Proc. of ACM MM*.
[58] James MacQueen et al. 1967. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Vol. 1. Oakland, CA, USA, 281–297.
[59] Aleix M Martinez and Avinash C Kak. 2001. Pca versus lda. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 23, 2 (2001), 228–233.
[60] Michael J May. 2022. Donag: Generating Efficient Patches and Diffs for Compressed Archives. *ACM Transactions on Storage* 18, 3 (2022), 1–41.
[61] Russ Mckendrick. [n. d.]. Migrating my Docker images to the GitHub Container Registry. https://medium.com/media-glasses/migrating-my-docker-images-to-the-github-container-registry-9f304ccf0aaa.
[62] Saeid Mofrad, Fengwei Zhang, Shiyong Lu, and Weidong Shi. 2018. A comparison study of intel SGX and AMD memory encryption technology. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. 1–8.
[63] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. 2001. A low-bandwidth network file system. In *Proc. of ACM SOSP*.
[64] Mohammad Norouzi, David J Fleet, and Russ R Salakhutdinov. 2012. Hamming distance metric learning. *NeurIPS* (2012).
[65] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. *Acm Sigmod Record* 22, 2 (1993), 297–306.
[66] Savan Oswal, Anjali Singh, and Kirthi Kumari. 2016. Deflate compression algorithm. *International Journal of Engineering Research and General Science* 4, 1 (2016), 430–436.
[67] Jisung Park, Jeonggyun Kim, Yeseong Kim, Sungjin Lee, and Onur Mutlu. 2022. DeepSketch: A New Machine Learning-Based Reference Search Technique for Post-Deduplication Delta Compression. In *Proc. of USENIX FAST*.
[68] Michael O Rabin. 1981. Fingerprinting by random polynomials. *Technical report* (1981).
[69] Yanjing Ren, Jingwei Li, Zuoru Yang, Patrick PC Lee, and Xiaosong Zhang. 2021. Accelerating Encrypted Deduplication via SGX. In *Proc. of USENIX ATC*.
[70] Vincent Rijmen and Joan Daemen. 2001. Advanced Encryption Standard. *FIPS* 19 (2001), 22.
[71] Ruslan Salakhutdinov and Geoffrey Hinton. 2009. Semantic hashing. *International Journal of Approximate Reasoning* 50, 7 (2009), 969–978.
[72] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. *ACM Transactions on Database Systems* 42, 3 (2017), 1–21.
[73] AMD Sev-Snp. 2020. Strengthening VM isolation with integrity protection and more. *White Paper, January* 53 (2020), 1450–1465.
[74] Philip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu. 2012. WAN optimized replication of backup datasets using stream-informed delta compression. In *Proc. of USENIX FAST*.
[75] Sonatype. [n. d.]. Nexus Repository Helps Developers Overcome New Docker Hub Rate Limits. https://blog.sonatype.com/nexus-repository-helps-developers-overcome-new-docker-hub-rate-limits.
[76] Mark W Storer, Kevin Greenan, Darrell DE Long, and Ethan L Miller. 2008. Secure Data Deduplication. In *Proc. of ACM international workshop on Storage security and survivability*.
[77] Tong Sun, Bowen Jiang, Lewei Jin, Wenzhao Zhang, Yi Gao, Zhendong Li, and Wei Dong. 2024. Understanding Differencing Algorithms for Mobile Application Updates. *IEEE Transactions on Mobile Computing* (2024).
[78] Rong-Cheng Tu, Xianling Mao, and Wei Wei. 2020. MLS3RDUH: Deep Unsupervised Hashing via Manifold based Local Semantic Similarity Structure Reconstructing.. In *Proc. of IJCAI*.

[79] Suzhen Wu, Zhanhong Tu, Zuocheng Wang, Zhirong Shen, and Bo Mao. 2021. When delta sync meets message-locked encryption: A feature-based delta sync scheme for encrypted cloud storage. In *Proc. of IEEE ICDCS*.

[80] Suzhen Wu, Zhanhong Tu, Yuxuan Zhou, Zuocheng Wang, Zhirong Shen, Wei Chen, Wei Wang, Weichun Wang, and Bo Mao. 2023. FASTSync: A FAST Delta Sync Scheme for Encrypted Cloud Storage in High-bandwidth Network Environments. *ACM Transactions on Storage* 19, 4 (2023), 1–22.

[81] Wen Xia, Chunguang Li, Hong Jiang, Dan Feng, Yu Hua, Leihua Qin, and Yucheng Zhang. 2015. Edelta: A Word-Enlarging Based Fast Delta Compression Approach. In *Proc. of USENIX HotStorage*.

[82] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. 2016. FastCDC: A Fast and Efficient Content-Defined Chunking Approach for Data Deduplication. In *Proc. of USENIX ATC*.

[83] Erkun Yang, Tongliang Liu, Cheng Deng, Wei Liu, and Dacheng Tao. 2019. Distillhash: Unsupervised deep hashing by distilling data pairs. In *Proc. of IEEE/CVF CVPR*.

[84] Miin-Shen Yang, Chien-Yo Lai, and Chih-Ying Lin. 2012. A robust EM clustering algorithm for Gaussian mixture models. *Pattern Recognition* 45, 11 (2012), 3950–3961.

[85] Zuoru Yang, Jingwei Li, and Patrick PC Lee. 2022. Secure and Lightweight Deduplicated Storage via Shielded Deduplication-Before-Encryption. In *Proc. of USENIX ATC 22*.

[86] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1996. BIRCH: an efficient data clustering method for very large databases. *ACM SIGMOD* 25, 2 (1996), 103–114.

[87] Yucheng Zhang, Wen Xia, Dan Feng, Hong Jiang, Yu Hua, and Qiang Wang. 2019. Finesse: Fine-Grained Feature Locality based Fast Resemblance Detection for Post-Deduplication Delta Compression. In *Proc. of USENIX FAST*.

[88] Nannan Zhao. [n. d.]. DupHunter. https://github.com/nnzhaocs/DupHunter.

[89] Nannan Zhao, Hadeel Albahar, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Lukas Rupprecht, Ali Anwar, and Ali R Butt. 2020. DupHunter: Flexible High-Performance Deduplication for Docker Registries. In *Proc. of USENIX ATC*.

[90] Nannan Zhao, Muhui Lin, Hadeel Albahar, Arnab K Paul, Zhijie Huan, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Ali Anwar, et al. 2024. An End-to-end High-performance Deduplication Scheme for Docker Registries and Docker Container Storage Systems. *ACM Transactions on Storage* 20, 3 (2024), 1–35.

[91] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S Warke, Mohamed Mohamed, and Ali R Butt. 2019. Large-scale Analysis of the Docker Hub Dataset. In *Proc. of IEEE CLUSTER*.

[92] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343.