S2Dedup: SGX-enabled Secure Deduplication

Mariana Miranda[1], Tânia Esteves[1], Bernardo Portela[2], João Paulo[1]
[1]*HASLab INESC TEC & U.Minho* [2]*NOVA LINCS & U.Porto*

Abstract. Secure deduplication allows removing duplicate content at third-party storage services while preserving the privacy of users' data. However, current solutions are built with strict designs that cannot be adapted to storage service and applications with different security and performance requirements.
We present S2Dedup, a trusted hardware-based privacy-preserving deduplication system designed to support multiple security schemes that enable different levels of performance, security guarantees and space savings. An in-depth evaluation shows these trade-offs for the distinct Intel SGX-based secure schemes supported by our prototype. Moreover, we propose a novel Epoch and Exact Frequency scheme that prevents frequency analysis leakage attacks present in current deterministic approaches for secure deduplication while maintaining similar performance and space savings to state-of-the-art approaches.

1 Introduction

A recurrent behaviour noticeable in storage services is that identical data is being stored repeatedly while consuming unnecessary space. Deduplication strives at eliminating these redundant copies to save storage space and costs. Indeed, cloud storage providers, like Dropbox and Google Drive, employ deduplication to eliminate redundant data stored by their users [11, 27, 38], while studies show that deduplication can reduce stored data by up to 83% and 68% in backup and primary storage systems, respectively [34]. However, this approach is only feasible when security is not a central concern, as revealing cross-user duplicates to untrusted servers has been shown to expose privacy vulnerabilities [25, 45].

To avoid privacy breaches, standard practices suggest that users should encrypt their data before outsourcing it to third-party storage services. However, this leads to the scenario where identical data, owned by different users, results in ciphertexts with distinct content, thus making it impossible to apply deduplication. To enable cross-user deduplication over encrypted data, the conventional approach is to employ convergent encryption (CE) [21]. Namely, data is encrypted at the users' premises with a deterministic encryption scheme in which the secret key is derived from the data content itself. This means that identical content encrypted by different users will generate matching ciphertexts, thus enabling deduplication. However, by revealing matching ciphertexts, these schemes provide considerably less security guarantees than standard encryption schemes, which potentially hinders their usability in scenarios handling sensitive data [32].

Recently, there has been an emergence of hardware-assisted security technologies (*e.g.,* Intel SGX [18], ARM TrustZone [12]). The trusted execution environments provided by these can be leveraged to aid the process of secure deduplication, as it allows for sensitive data to be handled in its original form (*i.e.*, plaintext) at an untrusted storage server. However, the applicability and advantages of these technologies are still vaguely explored for deduplication [20].

This paper presents S2Dedup, a secure deduplication system that leverages Intel SGX to enable cross-user privacy-preserving deduplication at third-party storage services. Unlike previous work [16, 20, 27, 32], S2Dedup aims at: i) enabling multiple secure schemes that can be adapted to the performance and security requirements of different applications; ii) designing a deduplication scheme with stronger security guarantees than deterministic ones; and iii) avoiding the network performance overhead imposed by auxiliary trusted remote servers for performing secure deduplication. Namely, the paper provides the following contributions:

a) Design and Prototype. The design and implementation of a secure deduplication system that takes advantage of the security functionalities provided by trusted hardware (Intel SGX), and resorts to state-of-the-art frameworks to develop efficient user space storage solutions, namely SPDK [8, 9].

b) Secure Deduplication Schemes. A modular and extensible design that allows straightforward integration of state-of-the-art and novel secure deduplication schemes within S2Dedup. This enables tailored deployments for applications with distinct security and performance requirements.

c) Epoch and Exact Frequency Scheme. A novel secure scheme that combines the concept of deduplication epochs with the idea of limiting the number of duplicates per unique chunk. By masking the frequency of duplicate copies, this solution leaks less sensitive information to an adversary.

d) Experimental Evaluation. An extensive evaluation of S2Dedup open-source[1] prototype, including more than 300 hours of experiments and resorting to synthetic and real trace workloads, in which S2Dedup's *Epoch and Exact Frequency* scheme is compared against state-of-the-art secure deduplication solutions [20, 32]. Results show that S2Dedup enables more robust security techniques while maintaining similar deduplication effectiveness and performance.

2 Background and Related Work

This section overviews Intel SGX and discusses relevant related work for secure deduplication.

2.1 Intel Software Guard Extensions (SGX)

Intel SGX [5, 18], provides a set of instructions in recent Intel's processors that ensure integrity and confidentiality. This is achieved by assuming that any layer in the computer's software stack (firmware, hypervisor, OS) is potentially malicious, thus SGX only trusts the CPU's microcode and a few privileged containers, known as enclaves. Enclaves are private regions of memory whose contents are protected and unable to be accessed by any process outside of it, including those running at higher privilege levels. Also, computational outputs can be remotely attested with respect to the code of the enclave that produced them. This enables external clients to establish secure channels between specific enclaves, ensuring trustworthy executions in untrusted environments.

In this paper, Intel SGX was picked over other alternatives (*e.g.,* ARM TrustZone [12]) due to its reliable computing and security features, as well as, its wide usage across both academia [19, 23, 26, 29, 40, 49] and industry [1, 2].

2.2 Secure Deduplication

Deduplication is a widely used technique that enables the identification and removal of duplicate data across files or blocks stored by distinct users and at different times [35].

Briefly, data being written by users is partitioned into chunks (*e.g.*, whole files, blocks) at either the client or storage server premises. Then, these chunks are matched at the storage server in order to find duplicates. This is usually achieved by hashing[2] the content of chunks and using a metadata index for efficiently finding previously stored chunks with the same content. If the content of a chunk being written was already stored, then additional metadata structures are used to create logical pointers to this data, thus avoiding the storage of duplicates.

The traditional approach to enable cross-user deduplication over encrypted data is to apply, at the client premises, a deterministic *Convergent Encryption* (CE) scheme that derives the key used to encrypt the data from the data content itself, thus ensuring that identical plaintexts will result in matching ciphertexts. The *Message-Locked Encryption* (MLE) [16] scheme follows this premise and provides the foundation for numerous other approaches [15, 17, 21]. As MLE is susceptible to brute-force attacks, DupLESS (*Duplicateless Encryption for Simple Storage*) [27] proposed a scheme that converts a predictable message into an unpredictable one with the aid of an external key server (KS).

The previous schemes enable attackers at the untrusted server to identify if a given block is a duplicate or not, thus being susceptible to information leakage (*e.g.,* revealing potential associations between users and the data these share) [31]. This leakage can be reduced by performing deduplication in epochs, which ensures that an adversary can only infer duplicates within the same epoch [20]. Deduplication with epochs requires shared state that must be synchronized across all clients. CE is not stateful, and thus must be adapted to depend on an external key server, a secure proxy or employ trusted hardware on the storage server to control the change of an epoch. H. Dang and E. C. Chien [20] achieved secure deduplication between epochs by relying on the use of trusted hardware and reliable network proxies. However, the proxies add extra computation and network operations in the critical I/O path, thus decreasing storage performance.

[1] `https://github.com/mmm97/S2Dedup`

[2] When a strong cryptographic hash function is being used (*e.g.*, SHA-256), duplicate content can be identified by comparing the hash sums of two chunks, as the probability of hash collisions has been shown to be negligible even for Petabyte-scale storage systems [35, 45].

While also requiring a remote trusted proxy, TED [32] avoids leaking the frequency of duplicates by limiting the number of copies per unique chunk. This solution masks the real count for chunks with a large amount of duplicates, but reveals the number of copies for unique chunks below a certain threshold of duplicates. Also, the proposed estimated frequency counter can lead to the speculation that a block has a higher number of duplicates than in reality, which can lead to lower deduplication gains.

To the best of our knowledge, S2Dedup is the first modular solution supporting multiple secure deduplication schemes, each enabling different trade-offs in terms of security, performance and space savings. By leveraging trusted hardware, S2Dedup schemes do not require a remote trusted proxy, thus avoiding costly network calls at the critical storage path. Also, our novel *Epoch and Exact Frequency* scheme combines the idea of limiting the number of duplicates per unique chunk with deduplication epochs, thus providing more robust security guarantees while maintaining comparable performance and deduplication effectiveness to state-of-the-art solutions.

3 S2Dedup Overview

S2Dedup leverages trusted hardware technologies to enable cross-user privacy-preserving deduplication at third-party storage providers. To be adaptable to the performance, space savings and security requirements real-world use cases entail, S2Dedup supports multiple secure deduplication schemes.

3.1 Threat Model

Our design considers two main trusted environments: the client applications; and the trusted hardware running security-sensitive operations. S2Dedup protects clients' data against two main adversaries: a network adversary that eavesdrops the communication between client and server; and a strictly stronger adversary that gains access to the server. The first can observe the messages exchanged between clients and the trusted server. The latter can observe access patterns of the storage service and the management of deduplication data and metadata. This is particularly relevant, as these accesses allow for duplicate identification.

Furthermore, our adversary is parametrised by the duration of server-side corruption. This is a common approach in cryptography to consider realistic threats, in which a specific resource becomes maliciously controlled for a finite amount of time [50]. In turn, our secure deduplication system provides varying levels of guarantees, depending on the period during which the server is assumed to be vulnerable.

3.2 Components and Flow of Requests

S2Dedup architecture (Figure 1) assumes a server that persists data from several clients, and is equipped with trusted hardware technology. The trusted hardware module plays the role of a trust anchor within an untrusted environment. It manages the cryptographic material, ensuring that user data can be processed efficiently while providing confidentiality and integrity guarantees. S2Dedup assumes a brief bootstrap stage, where a secure channel is initialized between the client and the remote enclave. This will entail a message exchange protocol that, upon completion, establishes a symmetric key between the client and the remote S2Dedup enclave. Instrumenting enclave code in this way is a common requirement for these systems and has been shown to be achievable with minimal performance overhead [14, 40].

S2Dedup adopts general design principles similar to other academic and production systems [35, 45]. Namely, it follows an inline approach to ensure that duplicates are eliminated before being stored persistently. Also, duplicates are found at the fixed-size block granularity (*e.g.*, 4KiB chunks), to achieve a good trade-off between space efficiency and computational complexity. Finally, hash calculation and deduplication are performed exclusively at the server-side. Although this approach does not allow reaping the network bandwidth space savings of client-side deduplication, it makes it easier to support standard implementations and protocols at client premises while enabling stronger secure deduplication schemes (*e.g.*, cross-user deduplication with epochs). Indeed, any alteration in client-side behaviour towards deduplication reveals information regarding duplicate identification [13]. In more detail, the proposed solution works as follows.

a) Client: When a client wishes to write data to a remote storage server, it begins the process by encrypting it with its own key using a standard probabilistic encryption scheme (Figure 1-❶) to protect against attackers eavesdropping the network channels and targeting the untrusted server. Then, the encrypted data is sent over the network through

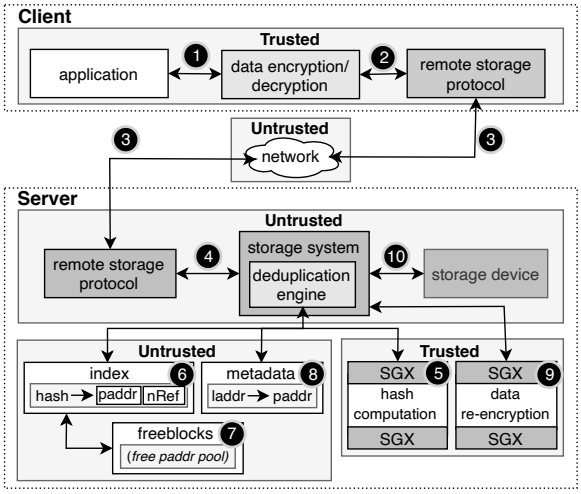


Fig. 1: S2Dedup deduplication architecture.

a standard remote storage protocol (*e.g.,* iSCSI) (Figure 1-❷❸). Note that each client request may contain several fixed-size blocks, which are only partitioned at the server side. Once the request reaches the remote server, the client's encrypted data is forwarded to the storage system (Figure 1-❹).

b) Deduplication Engine: The storage system integrates a deduplication engine that partitions and analyzes the request's data, block by block, in order to detect duplicates. Thus, the deduplication engine must calculate a cryptographic hash digest for each block (Figure 1-❺) so that it can compare it with the hashes of previously stored blocks. However, at this point of the execution, the content of each block is protected via a standard encryption scheme, which does not allow finding any duplicates. Indeed, the hash calculation must be done over the block contents, which should not be observed by the server. Instead, we rely on the trusted enclave to perform the hash computation step.

c) Index: Once a block's hash is calculated, the deduplication index is accessed (Figure 1-❻) to verify if the block being written is or not duplicated. Note that only the hash is disclosed to the index at the untrusted server. The contents of the block are never revealed outside of the trusted enclave.

In our design it is important to understand the difference between logical (*laddrs*) and physical (*paddrs*) addresses. As clients use the remote storage service as a standard block-device (*e.g.*, iSCSI device), their requests must specify the block-device offset (*laddr*) where new content (blocks) is going to be read or written. Then, the deduplication engine transparently eliminates duplicate blocks, storing only a single copy at a given *paddr* of the storage device. Redundancy is eliminated by pointing multiple *laddrs* to the same *paddr*.

A duplicate block was previously stored at the device if an identical content hash is found at the index. Otherwise, if a match was not found, a physical address (*paddr*) is requested from the freeblocks component (Figure 1-❼) and the index is updated with the corresponding hash sum and *paddr*. The freeblocks component is therefore responsible for tracking the *paddrs* of unused blocks at the server's storage medium.

d) Metadata: After collecting the *paddr* returned from the index (a duplicate was found) or from the freeblocks component (unique content is being written), the deduplication engine contacts the metadata component (Figure 1-❽). This component is responsible for mapping *laddrs* to *paddrs*. There are two possible flows when the metadata component is used: i) If the *laddr* is not pointing to any content at the storage device (*i.e.*, the application is writing a new block), the *paddr* returned by the index is inserted at the corresponding mapping; ii) if the *laddr* is pointing to content at the storage device (*i.e.*, the application is rewriting a previously stored block), the old *paddr* is updated with the new one return by the index. Note that for the case of delete operations, the corresponding *laddr* is updated to not point to any *paddr*.

e) Unused Blocks: When the metadata component completes its operation, and for the case when existing data was rewritten or deleted, the deduplication engine sends the old *paddr* to the index module to verify if that *paddr* is no longer being used (Figure 1-❻). Thus, the latter module also keeps track of the number of references (*nRef*) pointing to each index entry (*i.e.*, number of *laddr* sharing a given *paddr*). When a new duplicate is matched at the index, the number of references for the corresponding entry is increased by one. Contrarily, every time a *paddr* is deleted or

rewritten, the number of references is decreased by one. When the number of references reaches zero, the *paddr* is no longer being used and it is collected by the freeblocks component (Figure 1-❼).

f) Unique Blocks Storage: Once completed the deduplication process, if a block is unique then its content is re-encrypted using an *universal encryption key*, that is only accessible by the enclave (Figure 1-❾), and afterwards forwarded to the correct *paddr* at storage device (Figure 1-❿).

Re-encryption is needed to enable cross-client deduplication without requiring clients to share their encryption keys or derive them from the content being written, as in deterministic MLE [16]. Therefore, duplicate blocks shared among clients are protected using a single *universal encryption key* for data storage. Since we are dealing with private data, we must again resort to the trusted enclave for re-encryption. Our trusted component will decrypt client data, using the key exchanged during the bootstrap phase, and subsequently encrypt it with the *universal encryption key*.

The hash calculation and re-encryption steps are done in two separate enclave invocations. Alternatively, one could do both in a single enclave call, while always outputting to the untrusted environment the corresponding hash signature and encrypted block. Since deduplication is targeted towards storage workloads with a high percentage of duplicates, most write requests will not be persisted into the server's storage medium. Therefore, doing the re-encryption step for duplicate blocks, which will not be persisted, is unnecessarily increasing the computation done at the enclave as well as the data (encrypted blocks) being transferred from it to the untrusted environment. Similarly, due to performance reasons, our design ensures that enclaves are processing individual blocks instead of batches of blocks. This is important for trusted hardware solutions, such as Intel SGX, as several studies show that transferring large amounts of data to enclaves (*i.e.*, in the order of few MiB) exhibits worst performance than doing so with multiple smaller requests [24, 42, 43].

g) Read Operations: When a client retrieves data from the remote server, the metadata module (Figure 1-❽) is consulted to check where a given block (identified by a *laddr*) is stored at the storage device. The corresponding block (*paddr*) is read from the storage device (Figure 1-❿), decrypted with the *universal encryption key* and encrypted with the corresponding client's key (Figure 1-❾). Re-encryption is done at a secure enclave. Afterwards, the encrypted data is sent to the client (Figure 1-❹❸), where it is then decrypted and read by the application (Figure 1-❷❶).

4 Secure Deduplication

Our threat model intentionally considers an adversary more powerful than what might be necessary, which allows the provided S2Dedup secure schemes to respond to more powerful threats while still enabling the choice of deduplication scheme to be derived from real-world requirements.

4.1 Plain Security

The model presented in Section 3 constitutes the first security scheme offered by S2Dedup. Clients encrypt their data at trusted premises, with a standard encryption scheme and a private key, before sending it through the network. Once the encrypted data reaches the storage server, it is partitioned into blocks and processed with the aid of a trusted enclave.

The enclave obtains the client's key via the bootstrapping phase, which allows it to securely decrypt protected data and process it in plaintext. The block's hash digest can then be calculated, inside the enclave, using a specific cryptographic *hash key*, which is only accessible from within the enclave. The hash is then used by the deduplication engine, which operates outside of the enclave, to verify at the index if this is a duplicate block. In case that it is unique, data is encrypted again within the enclave, with the enclave's *universal encryption key* using a probabilistic encryption scheme.

Observe that deterministic encryption based systems (*e.g.,* CE) have two fundamental flaws when an adversary is given (even snapshot) access to the encrypted storage. The adversary is able to trivially see how many duplicate blocks are at the index, by simply checking for equality of ciphertexts; and it is able to test if any given block is in the storage, by encrypting it and checking for equality with the retrieved data. Due to the role played by the trusted enclave, none of these vulnerabilities exist in our system[3]. The security of this scheme collapses to the level of CE if we consider the adversary to be the server itself, since it can play the role of a client to test for duplicates, querying the system with values and checking if deduplication occurred. However, our scheme does not require establishing secure network channels (*e.g.*, TLS channels) to send encrypted blocks, as these are encrypted with a probabilistic scheme. Contrarily, CE requires secure channels to avoid disclosing duplicate blocks for adversaries eavesdropping the network.

[3] Our system also reveals duplicates if deduplication metadata is given to the attacker, but due to its relative size one can devise effective countermeasures that deter these types of threats.

4.2 Epoch Based

Our first alternative, supported by previous work [20], is an epoch based scheme. Its main idea is to establish a concrete temporal boundary (epoch) for deduplication, meaning that duplicate blocks stored in different epochs are considered to be different blocks for the deduplication engine.

The epochs are controlled by the enclave managing the client data and are defined by their specific *hash key*. The enclave is designed to automatically generate a new random *hash key*, depending on the criteria for epoch duration (*e.g.*, a pre-defined number of operations or time period). The main insight of this approach is that keyed hash functions over the same data with different keys (*i.e.,* in different epochs) will produce different digests, disabling deduplication.

As shown in Section 6.4, if storage workloads exhibit strong temporal locality, most duplicates will be written at the same epoch, thus not affecting significantly the achievable deduplication gain [35]. Nevertheless, the duration of an epoch can influence the achievable deduplication space savings and level of security. A smaller epoch offers higher security but leads to fewer duplicates being detected, while a larger epoch allows finding more duplicates at the cost of leaking more information about the number of duplicates of a given workload. Thus, our design leaves the duration of epochs as a configurable parameter that users can adjust.

We stress that the *hash key* is not used in the data re-encryption process. Indeed, adding epochs to the deduplication stage is seamless to the re-encryption process, as the enclave is simply encrypting with the *universal encryption key*, regardless of the epoch in which the blocks are stored.

This scheme achieves better security guarantees than the previous one. An adversary that gains access to the server is no longer capable of indefinitely testing for duplicates, as there is a limit of storage operations until the epoch changes.

4.3 Estimated Frequency Based

TED's solution [32] defines an upper bound for the number of duplicate copies that each stored chunk may have. This is achieved with the *Count-Min Sketch* algorithm, which uses a probabilistic data structure (matrix) to deduce the approximate number of copies (counters) per stored block. This structure is kept at a remote trusted server, which is consulted when a new block is being written. At the trusted server, it must first compute an hash sum over the block's plaintext in order to find how many copies of that block were previously stored (n). Then, it performs an integer division by threshold t and concatenates the result with the block's plaintext content, to calculate the hash signature used for the deduplication engine. This ensures that every set of t instances of writing a duplicate block produce different hash outputs, thus limiting the deduplication accordingly.

By keeping approximate counters, TED is able to reduce the memory footprint of the metadata structures holding these. However, this approach may overestimate the number of duplicates actually found for a given block. Thus, new hash sums may be generated before reaching exactly the expected threshold and miss some deduplication opportunities.

S2Dedup's *Estimated Frequency Based* scheme follows the same design as TED but uses trusted enclaves, instead of a remote trusted server, to do sensitive computations. This decision avoids extra network latency in the critical I/O path of write requests by leveraging the locally deployed enclave. Also, it enables a more fair comparison of this scheme with the remaining S2Dedup's secure deduplication approaches.

This scheme prevents access pattern attacks, which are prevalent on solutions that reveal duplicates [28]. These attacks succeed by establishing a relation between storage frequency (in our case) and statistical data from external sources, which is broken by the upper bound set by the frequency count metric. Thus, this scheme provides stronger security guarantees over the *Plain Security* one. However, observe that it does not prevent a malicious adversary from knowing exactly the number of references for chunks with low duplication counts, even if the corresponding duplicates were written across a large time period, which is addressed with the *Epoch Based* scheme. As such, the guarantees provided by these two schemes can be characterized as orthogonal – some threats are prevented by one, but not the other.

4.4 Epoch and Exact Frequency Based

We introduce a new solution that combines the security advantages of the two previous schemes: the epoch-based approach establishes a temporal boundary for duplicate detection, while the frequency-based approach allows masking the number of duplicates found within an epoch.

An upper bound for the amount of duplicates per block is ensured in the same way as for the *Estimated Frequency Based* scheme. The only difference is that this new scheme keeps an in-memory hash table at the secure enclave that maps blocks' hash sums to their exact number of copies (counters). By keeping the exact number of copies per block, we avoid the deduplication loss effects discussed in Section 4.3, which are a consequence of using estimated counters.

The secure enclave has limited memory space, which is not a problem for the estimated algorithm because it occupies a fixed memory space. However, the exact hash table used for this scheme increases in size with the number of new entries being added. Interestingly, the enclave's memory limitation can be useful for our solution as it can be used as an indicator to change deduplication epochs. Concretely, a new epoch can be introduced when the hash table structure reaches the maximum available memory at the enclave. When this occurs, we proceed to the next epoch, clearing the hash table information and generating a new *hash key*.

To sum up, by combining both techniques, this scheme is able to reach more robust security guarantees than the previous ones, while providing identical performance and space savings as the *Estimated Frequency Based* scheme (Section 6).

4.5 Security Analysis

Since deduplication is performed on the server-side, client data is sent encrypted using standard cryptographic techniques. This means that all our schemes provide semantic security guarantees against a network adversary.

When considering a server-side adversary, our *Epoch and Exact Frequency Based* scheme combines the strengths of the *Epoch Based* and *Estimated Frequency Based* schemes to provide two types of security guarantees: inter-epoch security and intra-epoch security. The first refers to blocks stored (by genuine clients or by the adversary) between different epochs. In this case, we have exactly semantic security: storage is done using probabilistic encryption of the original data, and metadata computation is done using different cryptographic hash keys. The second refers to blocks stored within the same epoch. In this case, we provide a security level superior than classical CE solutions. This is achieved by relying on a threshold t, which parametrizes the security of our system as follows. Hash computation considers a counter parameter that, combined with the block contents, fixes the input of the first t duplicate blocks, thus behaving exactly like a convergent encryption scheme. When the threshold is reached, the counter changes the hash input to a new value for the next t copies of the same block, thus behaving as deduplication of a different block, and the process repeats. This means that inference attacks, prevalent threats in systems that reveal equality via deduplication, must be performed within the period defined by epochs, and must necessarily require less duplicates than the fixed frequency value.

S2Dedup can be extended with alternative secure schemes resorting to techniques such as Oblivious RAM (ORAM) [41] to further prevent attackers from disclosing duplicate blocks. In this paper, we prioritise feasibility and avoid techniques, such as ORAM, that would take a considerable toll in the performance of our storage solution [39, 48].

5 Implementation

S2Dedup prototype is implemented in C and leverages both Intel SGX and the user space *Storage Performance Development Kit* (SPDK) [8, 9]. The latter enables the implementation of stackable virtual block devices, with user-defined logic, that follow a standard block-device interface.

5.1 Deduplication engine

Our secure deduplication engine is implemented as an SPDK virtual block device. It intercepts incoming block I/O requests, performs secure deduplication, with a fixed block-size specified by users, and only then forwards the requests to the NVMe block device or another virtual processing layer, depending on the targeted SPDK deployment. These requests will eventually reach the NVMe driver and storage device unless intermediate processing eliminates this need (*e.g.*, duplicate writes). Moreover, SPDK also provides a set of storage protocols that can be stacked over the block device abstraction layer. Of these, our work uses an iSCSI (*Internet Small Computer System Interface*) target implementation that enables clients to access the storage server remotely.

Data (re-)encryption operations at the engine are done with the NIST standardised AES-XTS block cipher mode, since it is a length-preserving scheme (i.e. the length of the ciphertext is the same as of the plaintext), and does not

apply chaining, thus supporting random access to encrypted data [22]. Besides using a key to encrypt the data, this cipher mode also relies on a tweak parameter to ensure ciphertexts' randomness for identical plaintext content.

Two alternatives were considered for the deduplication engine implementation, one that stores the index and metadata components in memory, by using GLib [6], and another that does it persistently, by resorting to the LevelDB key-value store [7]. The latter uses a partition of the server's storage device to persist LevelDB information. This decision was taken to observe experimentally (Section 6) the I/O performance impact of S2Dedup schemes in implementations that are not limited by the technology that makes them crash-tolerant.

In both implementations, the freeblocks component is implemented persistently by reserving a region of the server's storage device to store unused physical addresses. As each address only occupies 8 bytes, several entries can be stored in a relatively small amount of disk space. Also, we have implemented an in-memory cache of unused addresses to speedup operations at the freeblocks component.

The previous implementations consider the necessary mechanisms to provide support for concurrent requests.

5.2 Secure Schemes

The *Plain Security* scheme uses HMAC-SHA256 as the keyed cryptographic hash function. The hash key is generated during the system initialization. In the case of a system reset or failure, S2Dedup is still able to decrypt the data once the system is restarted as the *universal encryption key* is securely stored on disk via the SGX sealing mechanism.

We recall that our current prototype assumes a bootstrapping phase for establishing a secure channel between the server-side SGX enclave and the client in order to exchange the client's encryption key. This functionality is not implemented, as we are interested in analysing the server-side overhead of our secure deduplication schemes, and it would just be a matter of instantiating with one of many key exchange protocols for SGX [14, 33] with minimal performance overhead. This process only needs to be done once, namely when the client connects to the storage server, then multiple data storage and retrieval requests can be done efficiently without requiring the exchange of keys.

The *Epoch Based* scheme's implementation differs from the *Plain Security* one by introducing the concept of epochs, which entails an *hash key* that must be updated at every epoch. An epoch is changed when a certain time has passed, or when a limit of operations (*e.g.,* writes) is reached, which is configurable by users. When this criteria is met, the enclave simply generates a fresh key, and deletes the previous one.

The *Estimated Frequency Based* scheme follows the *Count-Min Sketch* implementation proposed by TED [32], and uses a matrix with 4 rows and 2^{20} counters per row.

The *Epoch and Exact Frequency Based* scheme maintains an in-memory hash table within the SGX enclave to keep track of the exact numbers of duplicates for each unique block. Since SGX memory is limited, the maximum number of hash table entries is restricted (to $approx$ 2 million) and, when this limit is reached, the hash table is cleared and a new epoch starts (new *hash key* is generated). Since all relevant metadata is recorded at the persistent index component, keeping this hash table in memory does not compromise S2Dedup's fault-tolerance. If the hash table is lost (*e.g.,* server reboots), a new epoch with an empty hash table will begin while just missing some potential deduplication opportunities.

5.3 Client Implementation

SPDK features an iSCSI client virtual block device that integrates *libiscsi* [4] and allows clients to connect and transfer data to a remote storage system (S2Dedup server) through the iSCSI protocol. This block device was extended to provide transparent data encryption with the AES-XTS block cipher mode. Also, to provide a standard block-device interface for client applications (*e.g.*, filesystem), an SPDK Linux NBD driver was stacked on top of the iSCSI client block device.

6 Evaluation

Our experiments validate two main questions: i) How do the different secure deduplication schemes affect I/O performance? and ii) What is their impact in terms of resource consumption (*i.e.*, RAM, CPU, network) and space savings?

6.1 Methodology

All conducted experiments included a benchmarking tool running at the client premises and writing or reading 4KiB blocks of data. Collected metrics include I/O throughput and latency, and deduplication space savings. Furthermore, the *dstat* [44] tool was used to collect the CPU, RAM, and network usage at the client and server nodes.

Experiments were repeated 3 times while the mean and standard deviation were used to summarize observed values. For each experiment, the page cache was cleaned and the prototype was re-deployed. The storage system was populated with an identical dataset before running read workloads.

Client and storage servers had the following specifications: a hexa-core 3.00 GHz CPU (Intel Core i5-9500), 16 GB DDR4 RAM, a 250GB Samsung NVMe SSD 970 EVO Plus, and a 10Gb/s network link. Servers ran Ubuntu Server 18.04.4 LTS with Linux kernel version 4.15.0-99-generic. We used the release of SPDK v20.04, and the 2.10.100.2 version of SGX SDK and Intel SGX Platform Software (PSW).

6.2 Workloads and Setups

Synthetic workloads were provided by DEDISbench (commit #0956b9d [3,37]), a disk I/O block-based benchmarking tool for deduplication systems that generates synthetic data mimicking realistic content distributions [36, 46, 47, 51]. DEDISbench allows evaluating systems with basic operations (reads or writes) and controls what access pattern is followed (sequential, uniform, or zipfian). The latter access pattern simulates a scenario where a small group of blocks is more accessed than the remaining ones. The benchmark allows specifying the number of concurrent processes and the content distribution to be generated. We chose two distributions with distinct percentages of redundancy: *dist_highperf* with 25% and *dist_kernels* with 72% of duplicate blocks.

Realistic workloads were provided by three real traces [10,30,36]. These traces were collected for three weeks from three production systems at the Florida International University (FIU) Computer Science department and contemplate different I/O workloads that consist of an email server (*mail* workload), a file server (*homes* workload), and a virtual machine running two web-servers (*web-vm* workload). Traces were replayed at different speedups to showcase how S2Dedup behaves under different I/O stressing conditions.

To assess the different secure schemes supported by S2Dedup, both in-memory and persistent metadata implementations (using 4 KiB as the deduplication block size) of the following setups were considered:

a) Baseline. This setup uses the S2Dedup prototype with both client encryption and server-side security mechanisms disabled, thus offering deduplication over plaintext data and without any in-place security measures.

b) Plain. This setup uses the *Plain Security* scheme.

c) Epoch. This setup uses the *Epoch Based* scheme. Deduplication epochs are changed at every 4 million writes ($approx$ 15 GiB of written data).

d) Estimated. This setup uses the *Estimated Frequency Based* secure scheme proposed by TED [32]. The number of duplicates per unique block is limited to 15.

e) Exact. This setup uses the novel *Epoch and Exact Frequency Based* secure scheme. The number of duplicates per unique block is also limited to 15. The number of hashtable entries at the enclave is limited to $approx$ 2 million.

6.3 Synthetic Experiments

Synthetic experiments ran for 20 minutes, or until the aggregated I/O totalled 64 GiB. Each test followed a sequential (seq), uniform (uni) or zipfian (zip) access pattern and used 4 concurrent processes. The obtained results are shown in Table 1. Cells are colored according to their relative performance overhead to the *Baseline* setup (darker is worse). Unless stated otherwise, the standard deviation for these results is always under 1.21% of the sample mean. Also, the deduplication space savings achieved by each setup are depicted in Figure 2a. We only include the deduplication ratio for the zipfian write experiments with persistent implementations since the results are similar across all experiments.

Results Analysis. Overall, when security is introduced (*Plain* setup), there is a decrease in performance due to the additional overhead of data encryption/decryption, doing additional computation at the enclave and transferring information to and from it. This is more noticeable for the in-memory implementation write workloads. Namely, for *dist_highperf* sequential write experiments, the *Plain* setup experienced a reduction of throughput to almost a quarter (103.89 MiB/s) when compared to the non-secure baseline setup (421.60 MiB/s). Notably, the overhead is more visible when the baseline setup offers higher performance. Indeed, for persistent implementations the differences are not as drastic, in which this secure setup can maintain the throughput at around 60%, compared to the baseline deployment.

Table 1: Synthetic workloads results.

Overhead
<50%
50-100%
100-200%
>200%

			Throughput (MiB/s)					Latency (ms)				
			Baseline	*Plain*	*Epoch*	*Estimated*	*Exact*	*Baseline*	*Plain*	*Epoch*	*Estimated*	*Exact*
dist_highperf	in-memory	seq-read	240.95	162.28	182.17	182.96	182.44	0.259	0.384	0.343	0.341	0.343
		uni-read	100.24	78.63	78.69	79.17	78.25	0.622	0.792	0.793	0.788	0.797
		zip-read	236.98	179.7	179.19	180.94	178.99	0.264	0.348	0.348	0.344	0.348
		seq-write	421.60	103.89	105.91	85.35	80.45	0.141	0.596	0.584	0.725	0.77
		uni-write	267.51	100.9	101.56	82.5	78.24	0.228	0.613	0.609	0.751	0.792
		zip-write	433.11	118.05	118.32	96.44	91.26	0.137	0.524	0.521	0.641	0.679
	persistent	seq-read	141.05	151.3	152.02	151.5	151.83	0.443	0.412	0.412	0.412	0.412
		uni-read	65.24	73.79	74.14	73.95	73.61	0.957	0.845	0.841	0.844	0.848
		zip-read	106.33	166.74	167.62	167.29	166.21	0.586	0.373	0.372	0.372	0.376
		seq-write	87.92	50.74	50.5	44.79	44.43	0.704	1.225	1.231	1.389	1.401
		uni-write	84.03	49.46	49.18	43.7	43.1	0.736	1.258	1.265	1.423	1.444
		zip-write	131.26	59.71	59.92	52.65	51.89	0.469	1.039	1.037	1.18	1.197
dist_kernels	in-memory	seq-read	486.59	183.94	183.47	184.21	182.31	0.128	0.34	0.34	0.34	0.344
		uni-read	102.67	78.88	78.93	79.27	78.2	0.608	0.792	0.791	0.787	0.797
		zip-read	247.69	182.1	181.89	182.65	180.36	0.252	0.342	0.343	0.34	0.344
		seq-write	430.22	148.09	134.46	95.13	89.76	0.14	0.416	0.457	0.649	0.689
		uni-write	270.21	143.7	127.5	91.47	87.36	0.224	0.428	0.483	0.677	0.708
		zip-write	445.70	166.16	147.11	108.5	102.87	0.133	0.368	0.419	0.568	0.601
	persistent	seq-read	245.82	156.02	151.77	151.78	150.62	0.254	0.4	0.412	0.412	0.415
		uni-read	76.06	74.44	73.89	74.1	73.56	0.82	0.839	0.844	0.843	0.848
		zip-read	137.97	170.6	168.56	168.55	167.98	0.452	0.364	0.368	0.369	0.372
		seq-write	111.51	69.91	62.74	49.35	49.47	0.555	0.888	0.99	1.261	1.257
		uni-write	105.11	66.27	59.91	48.6	47.89	0.587	0.936	1.037	1.279	1.299
		zip-write	151.07	77.5	70.75	58.38	57.08	0.407	0.8	0.877	1.063	1.088

For read experiments, the performance differences are less severe since these requests only require doing re-encryption operations at the SGX enclave, while write requests must do both re-encryption and hash computation at the enclave.

For all experiments, the performance overhead is similar across the *dist_highperf* and *dist_kernels* distributions and, as expected, the *Plain* setup achieves identical space savings to a baseline non-secure deduplication approach (Figure 2a).

For the *Epoch* setup, I/O throughput, latency, and space savings results are similar to the *Plain* configuration for the distribution *dist_highperf*, which presents a relatively small number of duplicates ($approx$ 17%). The same does not hold for distribution *dist_kernels*, which with the change of epochs suffered a reduction in the deduplication gain, approximately 15% when compared to *Plain* or *Baseline* setups. This decrease is most noticeable when the deduplication ratio is higher, and the workload does not exhibit temporal locality properties. Indeed, DEDISbench presents a worst-case scenario since duplicate content is written uniformly across time. The drop in the deduplication ratio meant more write operations to disk, which led to a slightly lower throughput across write workloads. Namely, the highest overhead was observed for zipfian writes that dropped from 166.16 MiB/s to 147.11 MiB/s.

When compared to the *Epoch* setup, the deduplication gain for the *Estimated* deployment dropped approximately 2% and 15% for the *dist_highperf* and *dist_kernels* workloads, respectively. This decrease is justified by the limits imposed on the number of duplicates per unique block. Again, this loss is translated into higher performance overhead for write requests. Sequential workloads present the highest drop in performance (20% for *dist_highperf* and 30% for *dist_kernels*).

Lastly, the *Exact* setup exhibits very similar performance values to the *Estimated* deployment. The most significant difference is visible at sequential write workloads (*dist_highperf*) and is less than 6%. On the other hand, for the *dist_kernels* workloads, this scheme increases space savings by almost 2%. As mentioned in Section 2, the *Estimated* approach can only ensure an approximate number of duplicates per unique block, hence losing some deduplication opportunities for blocks that have not yet reached the defined threshold (15 duplicates per block). Therefore, our *Exact* scheme can achieve similar performance and space savings even when adding deduplication epochs.

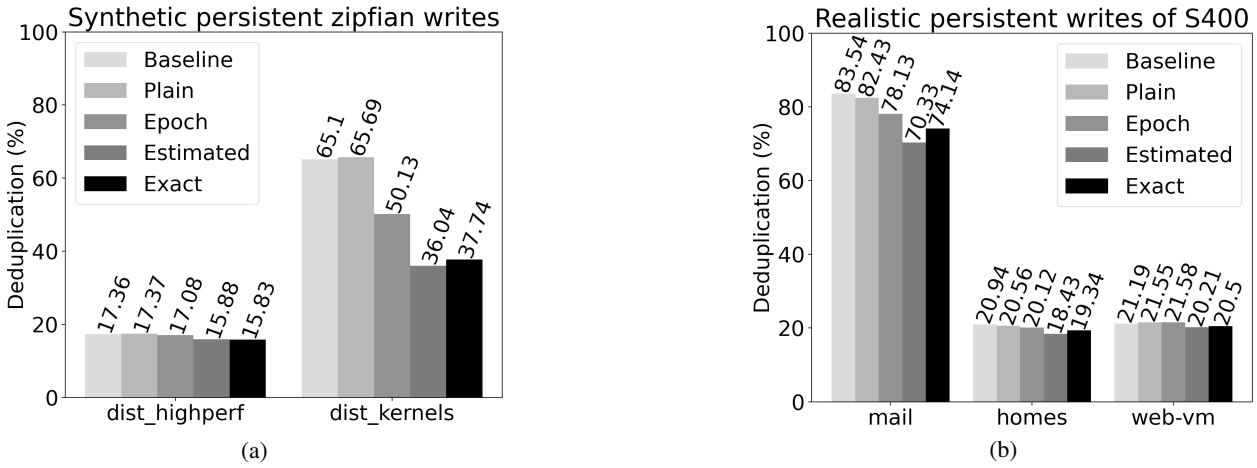


Fig. 2: Deduplication ratio results.

6.4 Realistic Experiments

S2Dedup prototype was evaluated with the - *mail*, *homes* and *web-vm* - traces described at Section 6.2. Each experiment lasted for 40 minutes. We devised a simple tracing tool, written in C, that replayed the I/O events described at the trace files with different speedups (S). Although the workloads perform a mix of read and write operations, at Table 2 we differentiate the results for each to enable a more insightful discussion. It follows the same colour scheme as the previous table. Also, we only show the results for S2Dedup persistent implementations, as the conclusions are identical for the in-memory ones. Figure 2b portrays deduplication space savings for the experiments with a speedup of x400. Results for other speedup values showed similar conclusions.

Results Analysis. When security is introduced at the *Plain* setup, the *mail* trace's read and write operations show a reduction on throughput for speedups of x200 (14% for reads and 8% for writes) and x400 (33% for reads and 29% for writes). This is justifiable because the peak workload saturation is reached at a speedup of 200x for this trace and secure setup. However, at other speedups (S1) and traces, the disk bandwidth peak saturation is not reached, thus the *Plain* setup does not present any throughput impact. Hence, we should redirect our attention to the latency results, in which is visible an overall increase for both read and write operations. This is expected given the extra work performed by the hash's computation and data re-encryption mechanisms within the SGX enclaves. Note that few latency results, especially for small speedups, show higher variability (marked with $*$), therefore we avoid drawing conclusions and comparing them. We suspect this variability to be caused by the conjunction of these workloads with the caching and scheduling mechanisms intrinsic to the operating system and hardware supporting the experiments.

With the introduction of epochs (*Epoch* setup), the space savings remained similar. Indeed, the highest difference is visible for the *mail* trace and is less than 5%. This behavior did not happen for the *dist_kernels* distribution at the synthetic tests, which experienced a 15% reduction of deduplication savings. This corroborates that temporal locality, typically present in real workloads, can attenuate the loss of deduplication space savings inherent to secure epochs. When it comes to performance, throughput and latency results are similar to the ones observed for the *Plain* setup.

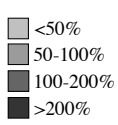
The *Estimated* setup shows the biggest differences in terms of space savings, mainly for the trace *mail* where space savings decrease by 18%. Again, this decrease is a result of limiting the number of duplicates per unique blocks. In terms of throughput and latency there is a higher performance penalty ($approx$ 13%) for the *mail* trace at the 200x and 400x speedups for both read and write operations.

The *Estimated* and *Exact* setups have very similar performance results, but it is perceptible a slight overall improvement in the latter's deduplication ratio. Namely, it reaches improvements of 4% for the *mail* trace.

6.5 Discussion

Resource usage across the different setups and experiments was similar. CPU usage was around 50% for the server and 18% for the client machines. Server's RAM usage for S2Dedup persistent implementations remained around 3.5 GB, while the in-memory implementations increased memory consumption to 8 GB, as a consequence of keeping additional metadata at RAM. Client's RAM was always below 4 GB.

Table 2: Realistic workloads results.



			Throughput (MiB/s)					Latency (ms)				
			Baseline	*Plain*	*Epoch*	*Estimated*	*Exact*	*Baseline*	*Plain*	*Epoch*	*Estimated*	*Exact*
mail	read	S1	2.24	2.24	2.24	2.25	2.24	55.71*	62.67*	83.53*	55.73*	41.79*
		S200	23.60	20.14	21.72	18.13	18.21	97.91	139.40	126.27	148.22	143.15
		S400	31.77	21.41	22.5	18.5	18.34	95.61	136.32	124.81	146.79	147.39
	write	S1	1.20	1.2	1.2	1.2	1.2	0.001	0.001	0.001	0.001	0.001
		S200	180.56	166.44	168.61	148.05	149.05	1.94	5.44	5.63	7.48	7.92
		S400	234.93	168.17	169.77	151.51	150.34	3.56	5.74	6.30	7.69	7.86
homes	read	S1	0.00	0.001	0.001	0.001	0.001	0.54	843.6*	0.573	422.1*	0.5
		S350	0.65	0.645	0.645	0.645	0.645	172.87	280.81	233.59	293.47	260.58
		S700	0.89	0.885	0.885	0.885	0.885	177.32	295.11	255.85	333.78	306.77
	write	S1	0.08	0.08	0.08	0.08	0.08	0.001	0.001	0.001	0.001	0.001
		S350	22.77	22.77	22.77	22.77	22.77	1.12	0.882	1.144	1.048	0.763
		S700	27.70	27.7	27.7	27.7	27.7	1.018	1.097	0.803	1.018	0.94
web-vm	read	S1	0.05	0.05	0.05	0.05	0.05	0.299	0.30	0.31	0.32	0.297
		S350	1.87	1.87	1.87	1.87	1.87	31.73	39.00	33.76	36.38	30.27
		S700	4.47	4.47	4.47	4.47	4.47	16.30	17.15	17.03	19.10	16.91
	write	S1	0.02	0.02	0.02	0.02	0.02	0.002	0.002	0.002	0.002	0.002
		S350	6.94	6.94	6.94	6.94	6.94	0.86	1.095	0.939	0.743	1.329
		S700	16.78	16.78	16.78	16.78	16.78	0.841	0.873	1.325	0.808	1.422

Results show that a simpler secure deduplication scheme (*Plain*), aided by Intel SGX, does not affect deduplication gain but may lead to performance loss, especially under stress I/O loads. By introducing epochs (*Epoch*), S2Dedup ensures stronger security guarantees at a small cost in terms of performance and space savings. As expected, the *Estimated* setup provides different security guarantees at the cost of reduced performance and space savings. Interestingly, when compared to the latter, our novel *Epoch and Exact Frequency Based* scheme offers more robust security guarantees while maintaining similar performance and deduplication gain.

7 Conclusion

This paper proposes S2Dedup, a secure deduplication system based on trusted hardware. By supporting multiple schemes, S2Dedup can offer tailored deployments for applications with distinct requirements in terms of security, performance, and space savings. Furthermore, we introduce a novel *Epoch and Exact Frequency* scheme that, when compared to state-of-the-art solutions, enables improved security without sacrificing storage performance and deduplication space-savings.

A prototype of S2Dedup, leveraging both Intel SGX and the SPDK framework, is evaluated under synthetic and real workloads. The experiments highlight the importance of knowing the trade-offs of different secure schemes when applied in practice. We believe that this knowledge, coupled with solutions that enable different security choices, is fundamental to push forward a wider usage of secure deduplication in third-party storage services, while promoting the privacy and security for individuals using such services.

As future work, it would be interesting to compare S2Dedup's schemes with other state-of-the-art solutions, that rely on CE or remote key servers [20, 32], to better pinpoint the performance trade-offs of using secure enclaves and network I/O calls.

Acknowledgments

Work funded by National Funds through the FCT—Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through project PAStor (UTA-EXPL/CA/0075/2019 - M. Miranda), project HADES

(PTDC/CCI-INF/31698/2017 - B. Portela), PhD grant (DFA/BD/5881/2020 - T. Esteves), and NOVA LINCS (UIDB/04516/2020 - B. Portela). We would like to thank our shepherd Jeanna Matthews and the anonymous reviewers for their insightful comments that helped us improve this paper.

References

1. A More Protected Cloud Environment: IBM Announces Cloud Data Guard Featuring Intel SGX. https://itpeernetwork.intel.com/ibm-cloud-data-guard-intel-sgx/#gs.oejhq1, 2017.
2. Azure confidential computing. https://azure.microsoft.com/en-us/blog/azure-confidential-computing, 2018.
3. DEDISbench. https://github.com/jtpaulo/dedisbench, 2020.
4. Libiscsi. https://github.com/sahlberg/libiscsi.git, 2020.
5. Intel Software Guard Extensions (Intel SGX). https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html, Accessed: 2019-09-19.
6. Glib. https://github.com/GNOME/glib.git, Accessed: 2019-11-26.
7. LevelDB. https://github.com/google/leveldb.git, Accessed: 2020-03-16.
8. Spdk github. https://github.com/spdk/spdk, Accessed: 2020-05-15.
9. Storage performance development kit. https://spdk.io/, Accessed: 2020-05-15.
10. FIU IODedup. http://iotta.snia.org/traces/391, Accessed: 2020-08-27.
11. K. Akhila, A. Ganesh, and C. Sunitha. A study on deduplication techniques over encrypted data. *Procedia Computer Science*, 87:38–43, 2016.
12. T. Alves. Trustzone: Integrated hardware and software security. *White paper*, 2004.
13. F. Armknecht, C. Boyd, G. T. Davies, K. Gjøsteen, and M. Toorani. Side channels in deduplication: trade-offs between leakage and efficiency. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 266–274. ACM, 2017.
14. R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A.-R. Sadeghi, G. Scerri, and B. Warinschi. Secure multiparty computation from sgx. In *International Conference on Financial Cryptography and Data Security*, pages 477–497. Springer, 2017.
15. M. Bellare and S. Keelveedhi. Interactive message-locked encryption and secure deduplication. In *IACR International Workshop on Public Key Cryptography*, pages 516–538. Springer, 2015.
16. M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 296–312. Springer, 2013.
17. R. Chen, Y. Mu, G. Yang, and F. Guo. Bl-mle: Block-level message-locked encryption for secure large file deduplication. *Information Forensics and Security, IEEE Transactions on*, 10:2643–2652, 12 2015.
18. V. Costan and S. Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.
19. H. Cui, H. Duan, Z. Qin, C. Wang, and Y. Zhou. Speed: Accelerating enclave applications via secure deduplication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1072–1082. IEEE, 2019.
20. H. Dang and E.-C. Chang. Privacy-preserving data deduplication on trusted processors. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 66–73. IEEE, 2017.
21. J. J. Douceur, A. Adya, B. Bolosky, D. R. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. Technical Report MSR-TR-2002-30, July 2002.
22. M. J. Dworkin. Recommendation for block cipher modes of operation: The xts-aes mode for confidentiality on storage devices. Technical report, 2010.
23. B. Fuhry, L. Hirschoff, S. Koesnadi, and F. Kerschbaum. Segshare: Secure group file sharing in the cloud using enclaves. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 476–488. IEEE, 2020.
24. D. Harnik, E. Tsfadia, D. Chen, and R. Kat. Securing the storage data path with sgx enclaves. *arXiv preprint arXiv:1806.10883*, 2018.
25. W. Hu, T. Yang, and J. N. Matthews. The good, the bad and the ugly of consumer cloud storage. *ACM SIGOPS Operating Systems Review*, 44(3):110–115, 2010.
26. T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Transactions on Computer Systems (TOCS)*, 35(4):13, 2018.
27. S. Keelveedhi, M. Bellare, and T. Ristenpart. Dupless: server-aided encryption for deduplicated storage. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 179–194, 2013.
28. G. Kellaris, G. Kollios, K. Nissim, and A. O'neill. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1340, 2016.
29. S. Kim, Y. Shin, J. Ha, T. Kim, and D. Han. A first step towards leveraging commodity trusted execution environments for network applications. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, page 7. ACM, 2015.
30. R. Koller and R. Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *ACM Transactions on Storage (TOS)*, 6(3):1–26, 2010.

31. J. Li, C. Qin, P. P. Lee, and X. Zhang. Information leakage in encrypted deduplication via frequency analysis. In *2017 47th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 1–12. IEEE, 2017.
32. J. Li, Z. Yang, Y. Ren, P. P. Lee, and X. Zhang. Balancing storage efficiency and data confidentiality with tunable encrypted deduplication. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.
33. T. Machida, D. Yamamoto, I. Morikawa, H. Kokubo, and H. Kojima. Poster: A novel framework for user-key provisioning to secure enclaves on intel sgx.
34. D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *ACM Transactions on Storage (ToS)*, 7(4):1–20, 2012.
35. J. Paulo and J. Pereira. A survey and classification of storage deduplication systems. *ACM Computing Surveys (CSUR)*, 47(1):11, 2014.
36. J. Paulo and J. Pereira. Efficient deduplication in a distributed primary storage infrastructure. *ACM Transactions on Storage (TOS)*, 12(4):1–35, 2016.
37. J. Paulo, P. Reis, J. Pereira, and A. Sousa. Dedisbench: A benchmark for deduplicated storage systems. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 584–601. Springer, 2012.
38. Z. Pooranian, K.-C. Chen, C.-M. Yu, and M. Conti. Rare: Defeating side channels based on data-deduplication in cloud storage. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 444–449. IEEE, 2018.
39. J. Rakshit and K. Mohanram. Leo: Low overhead encryption oram for non-volatile memories. *IEEE Computer Architecture Letters*, 17(2):100–104, 2018.
40. F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54. IEEE, 2015.
41. E. Stefanov, M. V. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. *Journal of the ACM (JACM)*, 65(4):1–26, 2018.
42. B. Vavala, N. Neves, and P. Steenkiste. Secure tera-scale data crunching with a small tcb. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 169–180. IEEE, 2017.
43. O. Weisse, V. Bertacco, and T. Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. *ACM SIGARCH Computer Architecture News*, 45(2):81–93, 2017.
44. D. Wieers. Dstat: Versatile resource statistics tool. *http://dag.wiee.rs/home-made/dstat*.
45. W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.
46. Q. Yang, R. Jin, and M. Zhao. Smartdedup: Optimizing deduplication for resource-constrained devices. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 633–646, Renton, WA, July 2019. USENIX Association.
47. H. Yu, X. Zhang, W. Huang, and W. Zheng. Pdfs: Partially dedupped file system for primary workloads. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):863–876, 2017.
48. X. Zhang, G. Sun, C. Zhang, W. Zhang, Y. Liang, T. Wang, Y. Chen, and J. Di. Fork path: improving efficiency of oram by removing redundant memory accesses. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 102–114. IEEE, 2015.
49. W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 283–298, 2017.
50. L. Zhou, F. B. Schneider, and R. Van Renesse. Apss: Proactive secret sharing in asynchronous systems. *ACM transactions on information and system security (TISSEC)*, 8(3):259–286, 2005.
51. Y. Zhou, Y. Deng, L. T. Yang, R. Yang, and L. Si. Ldfs: A low latency in-line data deduplication file system. *IEEE Access*, 6:15743–15753, 2018.